Introduction to computing with finite difference methods

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Aug 11, 2014

Contents

Finite difference methods for partial differential equations (PDEs) e
range of concepts and tools that can be introduced and illustrated in the
of simple ordinary differential equation (ODE) examples. This is what
the present document. By first working with ODEs, we keep the mathe
problems to be solved as simple as possible (but no simpler), thereby
full focus on understanding the key concepts and tools. The choice o

ı the forthcoming treatment of ODEs is therefore solely dominated by what arries over to numerical methods for PDEs.

Theory and practice are primarily illustrated by solving the very simple DE $u' = -au$, $u(0) = I$, where $a > 0$ is a constant, but we also address the eneralized problem $u' = -a(t)u + b(t)$ and the nonlinear problem $u' = f(u, t)$. he following topics are introduced:

- How to think when constructing finite difference methods, with special focus on the Forward Euler, Backward Euler, and Crank-Nicolson (midpoint) schemes
- How to formulate a computational algorithm and translate it into Python code
- How to make curve plots of the solutions
- How to compute numerical errors
- How to compute convergence rates
- How to verify an implementation and automate verification through nose tests in Python
- How to structure code in terms of functions, classes, and modules
- How to work with Python concepts such as arrays, lists, dictionaries, lambda functions, functions in functions (closures), doctests, unit tests, command-line interfaces, graphical user interfaces
- How to perform array computing and understand the difference from scalar computing
- How to conduct and automate large-scale numerical experiments
- How to generate scientific reports
- How to uncover numerical artifacts in the computed solution
- How to analyze the numerical schemes mathematically to understand why artifacts occur
- How to derive mathematical expressions for various measures of the error in numerical methods, frequently by using the `sympy` software for symbolic computation
- Introduce concepts such as finite difference operators, mesh (grid), mesh functions, stability, truncation error, consistency, and convergence
- Present additional methods for the general nonlinear ODE $u' = f(u, t)$, which is either a scalar ODE or a system of ODEs
- How to access professional packages for solving ODEs

- How the model equation $u' = -au$ arises in a wide range of pheno physics, biology, and finance

The exposition in a nutshell.

Everything we cover is put into a practical, hands-on context. All mathe ics is translated into working computing codes, and all the mathema theory of finite difference methods presented here is motivated fr strong need to understand strange behavior of programs. Two fundam questions saturate the text:

- How to we solve a differential equation problem and produce num
- How to we trust the answer?

1 Finite difference methods

Goal.

We explain the basic ideas of finite difference methods using a si ordinary differential equation $u' = -au$ as primary example. Emp is put on the reasoning when discretizing the problem and introdu of key concepts such as mesh, mesh function, finite difference appro tions, averaging in a mesh, derivation of algorithms, and discrete ope notation.

1.1 A basic model for exponential decay

Our model problem is perhaps the simplest ordinary differential equation

$$u'(t) = -au(t),$$

Here, $a > 0$ is a constant and $u'(t)$ means differentiation with respect t This type of equation arises in a number of widely different phenomen some quantity u undergoes exponential reduction. Examples include rad decay, population decay, investment decay, cooling of an object, pressu in the atmosphere, and retarded motion in fluids (for some of these a can be negative as well), see Section 8 for details and motivation. V chosen this particular ODE not only because its applications are relev even more because studying numerical solution methods for this simp gives important insight that can be reused in much more complicated s in particular when solving diffusion-type partial differential equations.

The analytical solution of the ODE is found by the method of separation of ariables, which results in

$$u(t) = Ce^{-at},$$

or any arbitrary constant C. To formulate a mathematical problem for which here is a unique solution, we need a condition to fix the value of C. This ondition is known as the *initial condition* and stated as $u(0) = I$. That is, we now the value I of u when the process starts at $t = 0$. The exact solution is hen $u(t) = Ie^{-at}$.

We seek the solution $u(t)$ of the ODE for $t \in (0, T]$. The point $t = 0$ is not ncluded since we know u here and assume that the equation governs u for $t > 0$. he complete ODE problem then reads: find $u(t)$ such that

$$u' = -au, \ t \in (0, T], \quad u(0) = I\,. \tag{1}$$

his is known as a *continuous problem* because the parameter t varies continuously om 0 to T. For each t we have a corresponding $u(t)$. There are hence infinitely any values of t and $u(t)$. The purpose of a numerical method is to formulate a orresponding *discrete* problem whose solution is characterized by a finite number f values, which can be computed in a finite number of steps on a computer.

.2 The Forward Euler scheme

olving an ODE like (1) by a finite difference method consists of the following our steps:

1. discretizing the domain,
2. fulfilling the equation at discrete time points,
3. replacing derivatives by finite differences,
4. formulating a recursive algorithm.

tep 1: Discretizing the domain. The time domain $[0, T]$ is represented y a finite number of $N_t + 1$ points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{2}$$

he collection of points $t_0, t_1, \ldots, t_{N_t}$ constitutes a *mesh* or *grid*. Often the esh points will be uniformly spaced in the domain $[0, T]$, which means that ne spacing $t_{n+1} - t_n$ is the same for all n. This spacing is often denoted by Δt, n this case $t_n = n\Delta t$.

We seek the solution u at the mesh points: $u(t_n)$, $n = 1, 2, \ldots, N_t$. Note that ^{0} is already known as I. A notational short-form for $u(t_n)$, which will be used xtensively, is u^n. More precisely, we let u^n be the *numerical approximation* to the

exact solution $u(t_n)$ at $t = t_n$. The numerical approximation is a *mesh f* here defined only at the mesh points. When we need to clearly dist between the numerical and the exact solution, we often place a sub on the exact solution, as in $u_e(t_n)$. Figure 1 shows the t_n and u_n pc $n = 0, 1, \ldots, N_t = 7$ as well as $u_e(t)$ as the dashed line. The goal of a nu method for ODEs is to compute the mesh function by solving a finit *algebraic equations* derived from the original ODE problem.

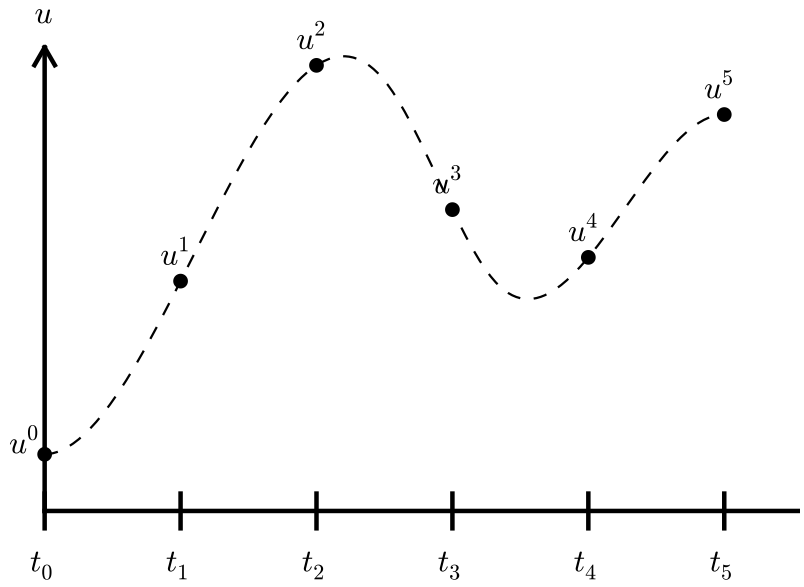
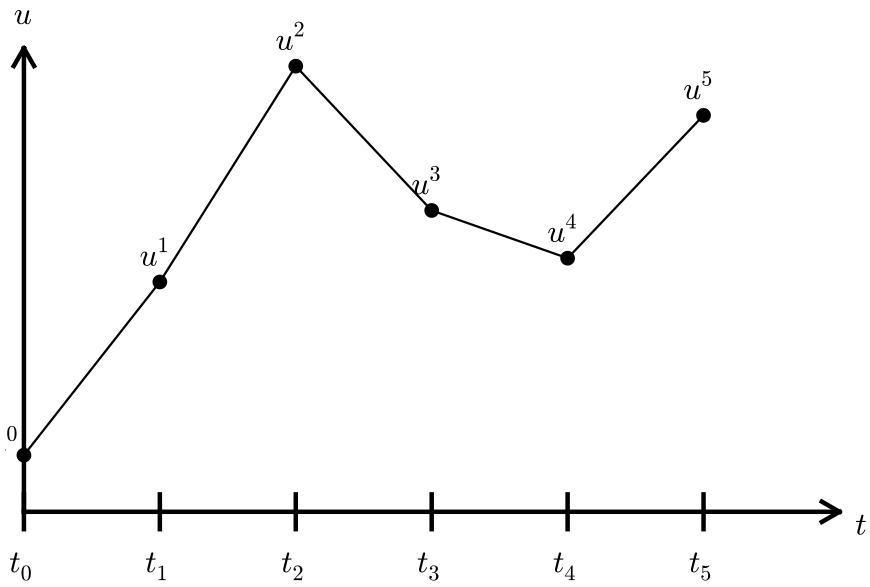


Figure 1: Time mesh with discrete solution values.

Since finite difference methods produce solutions at the mesh points is an open question what the solution is between the mesh points. One methods for interpolation to compute the value of u between mesh poir simplest (and most widely used) interpolation method is to assume that linearly between the mesh points, see Figure 2. Given u^n and u^{n+1}, the u at some $t \in [t_n, t_{n+1}]$ is by linear interpolation

$$u(t) \approx u^n + \frac{u^{n+1} - u^n}{t_{n+1} - t_n}(t - t_n)\,.$$

Step 2: Fulfilling the equation at discrete time points. The supposed to hold for all $t \in (0, T]$, i.e., at an infinite number of points. relax that requirement and require that the ODE is fulfilled at a finit discrete points in time. The mesh points $t_0, t_1, \ldots, t_{N_t}$ are a natural (the only) choice of points. The original ODE is then reduced to the fc N_t equations:



igure 2: Linear interpolation between the discrete solution values (dashed ırve is exact solution).

$$u'(t_n) = -au(t_n), \quad n = 0, \ldots, N_t\,. \tag{4}$$

tep 3: Replacing derivatives by finite differences. The next and most ssential step of the method is to replace the derivative u' by a finite differ- ıce approximation. Let us first try a one-sided difference approximation (see igure 3),

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{5}$$

ıserting this approximation in (4) results in

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^n, \quad n = 0, 1, \ldots, N_t - 1\,. \tag{6}$$

ater it will be absolutely clear that if we want to compute the solution up › time level N_t, we only need (4) to hold for $n = 0, \ldots, N_t - 1$ since (6) for $= N_t - 1$ creates an equation for the final value u^{N_t}.

Equation (6) is the discrete counterpart to the original ODE problem (1), nd often referred to as *finite difference scheme* or more generally as the *discrete quations* of the problem. The fundamental feature of these equations is that ıey are *algebraic* and can hence be straightforwardly solved to produce the ıesh function, i.e., the values of u at the mesh points (u^n, $n = 1, 2, \ldots, N_t$).

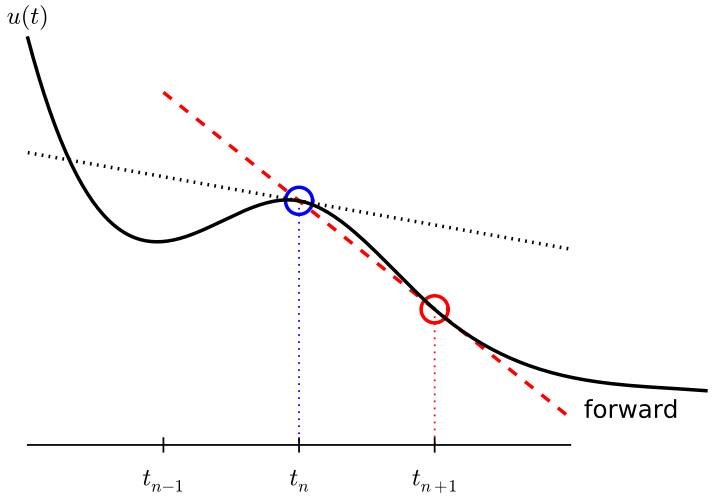


Figure 3: Illustration of a forward difference.

Step 4: Formulating a recursive algorithm. The final step is to the computational algorithm to be implemented in a program. The ke vation here is to realize that (6) can be used to compute u^{n+1} if u^n is Starting with $n = 0$, u^0 is known since $u^0 = u(0) = I$, and (6) gives an e for u^1. Knowing u^1, u^2 can be found from (6). In general, u^n in (6) assumed known, and then we can easily solve for the unknown u^{n+1}:

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n\,.$$

We shall refer to (7) as the Forward Euler (FE) scheme for our model p From a mathematical point of view, equations of the form (7) are kı *difference equations* since they express how differences in u, like $u^{n+1} - u^n$ with n. The finite difference method can be viewed as a method for tu differential equation into a difference equation.

Computation with (7) is straightforward:

$$\begin{aligned}
u_0 &= I,\\
u_1 &= u^0 - a(t_1 - t_0)u^0 = I(1 - a(t_1 - t_0)),\\
u_2 &= u^1 - a(t_2 - t_1)u^1 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1)),\\
u^3 &= u^2 - a(t_3 - t_2)u^2 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1))(1 - a(t_3 -
\end{aligned}$$

and so on until we reach u^{N_t}. Very often, $t_{n+1} - t_n$ is constant for ε we can introduce the common symbol Δt for the time step: $\Delta t = t_n$ $n = 0, 1, \ldots, N_t - 1$. Using a constant time step Δt in the above calc gives

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= I(1-a\Delta t),\\
u_2 &= I(1-a\Delta t)^2,\\
u^3 &= I(1-a\Delta t)^3,\\
&\vdots\\
u^{N_t} &= I(1-a\Delta t)^{N_t}\,.
\end{aligned}
$$

his means that we have found a closed formula for u^n, and there is no need to let a computer generate the sequence $u^1, u^2, u^3, \ldots$. However, finding such formula for u^n is possible only for a few very simple problems, so in general nite difference equations must be solved on a computer.

As the next sections will show, the scheme (7) is just one out of many lternative finite difference (and other) methods for the model problem (1).

.3 The Backward Euler scheme

here are several choices of difference approximations in step 3 of the finite ifference method as presented in the previous section. Another alternative is

$$
u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}}\,. \tag{8}
$$

ince this difference is based on going backward in time (t_{n-1}) for information, is known as the Backward Euler difference. Figure 4 explains the idea.

Inserting (8) in (4) yields the Backward Euler (BE) scheme:

$$
\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n\,. \tag{9}
$$

We assume, as explained under step 4 in Section 1.2, that we have computed $^0, u^1, \ldots, u^{n-1}$ such that (9) can be used to compute u^n. For direct similarity ith the Forward Euler scheme (7) we replace n by $n+1$ in (9) and solve for ıe unknown value u^{n+1}:

$$
u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)}u^n\,. \tag{10}
$$

.4 The Crank-Nicolson scheme

he finite difference approximations used to derive the schemes (7) and (10) are oth one-sided differences, known to be less accurate than central (or midpoint) ifferences. We shall now construct a central difference at $t_{n+1/2} = \frac{1}{2}(t_n + t_{n+1})$, r $t_{n+1/2} = (n+\frac{1}{2})\Delta t$ if the mesh spacing is uniform in time. The approximation eads

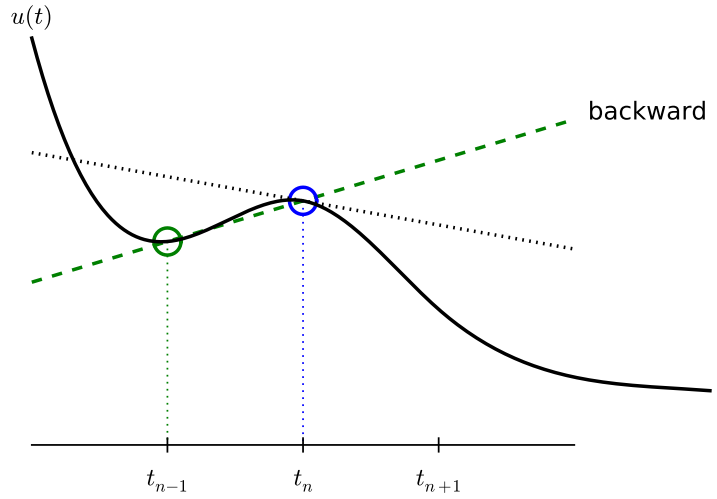


Figure 4: Illustration of a backward difference.

$$
u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,.
$$

Note that the fraction on the right-hand side is the same as for the I Euler approximation (5) and the Backward Euler approximation (8) replaced by $n+1$). The accuracy of this fraction as an approximatioı derivative of u depends on *where* we seek the derivative: in the cente interval $[t_n, t_{n+1}]$ or at the end points.

With the formula (11), where u' is evaluated at $t_{n+1/2}$, it is nat demand the ODE to be fulfilled at the time points *between* the mesh p

$$
u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}}), \quad n = 0, \ldots, N_t - 1\,.
$$

Using (11) in (12) results in

$$
\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^{n+\frac{1}{2}},
$$

where $u^{n+\frac{1}{2}}$ is a short form for $u(t_{n+\frac{1}{2}})$. The problem is that we aim to c u^n for integer n, implying that $u^{n+\frac{1}{2}}$ is not a quantity computed by our It must therefore be expressed by the quantities that we actually prod the numerical solution at the mesh points. One possibility is to appr $u^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighboring mesh p

$$
u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1})\,.
$$

Using (14) in (13) results in

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a\frac{1}{2}(u^n + u^{n+1})\,. \tag{15}$$

igure 5 sketches the geometric interpretation of such a centered difference.

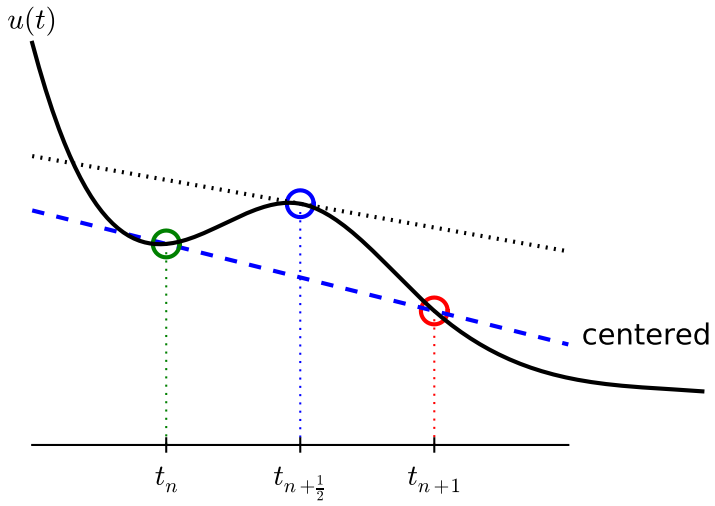


Figure 5: Illustration of a centered difference.

We assume that u^n is already computed so that u^{n+1} is the unknown, which e can solve for:

$$u^{n+1} = \frac{1-\frac{1}{2}a(t_{n+1}-t_n)}{1+\frac{1}{2}a(t_{n+1}-t_n)}u^n\,. \tag{16}$$

he finite difference scheme (16) is often called the Crank-Nicolson (CN) scheme r a midpoint or centered scheme.

.5 The unifying θ-rule

he Forward Euler, Backward Euler, and Crank-Nicolson schemes can be formu- ıted as one scheme with a varying parameter θ:

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a(\theta u^{n+1} + (1-\theta)u^n)\,. \tag{17}$$

Observe:

- $\theta = 0$ gives the Forward Euler scheme
- $\theta = 1$ gives the Backward Euler scheme, and
- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme.

- We may alternatively choose any other value of θ in $[0,1]$.

As before, u^n is considered known and u^{n+1} unknown, so we solve for th

$$u^{n+1} = \frac{1-(1-\theta)a(t_{n+1}-t_n)}{1+\theta a(t_{n+1}-t_n)}\,.$$

This scheme is known as the θ-rule, or alternatively written as the "the

Derivation.

We start with replacing u' by the fraction

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n},$$

in the Forward Euler, Backward Euler, and Crank-Nicolson schemes. we observe that the difference between the methods concerns which this fraction approximates the derivative. Or in other words, at v point we sample the ODE. So far this has been the end points o midpoint of $[t_n, t_{n+1}]$. However, we may choose any point $\tilde{t} \in [t_n,$ The difficulty is that evaluating the right-hand side $-au$ at an arbi point faces the same problem as in Section 1.4: the point value mu expressed by the discrete u quantities that we compute by the scheme u^n and u^{n+1}. Following the averaging idea from Section 1.4, the val u at an arbitrary point $\tilde{t}$ can be calculated as a *weighted average*, v generalizes the arithmetic mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. If we express $\tilde{t}$ as a weig average

$$t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n,$$

where $\theta \in [0,1]$ is the weighting factor, we can write

$$u(\tilde{t}) = u(\theta t_{n+1} + (1-\theta)t_n) \approx \theta u^{n+1} + (1-\theta)u^n\,.$$

We can now let the ODE hold at the point $\tilde{t} \in [t_n, t_{n+1}]$, approxi u' by the fraction $(u^{n+1}-u^n)/(t_{n+1}-t_n)$, and approximate the right- side $-au$ by the weighted average (19). The result is (17).

1.6 Constant time step

All schemes up to now have been formulated for a general non-uniform time: $t_0, t_1, \ldots, t_{N_t}$. Non-uniform meshes are highly relevant since one many points in regions where u varies rapidly, and save points in region u is slowly varying. This is the key idea of *adaptive* methods where the of the mesh points are determined as the computations proceed.

However, a uniformly distributed set of mesh points is very comm sufficient for many applications. It therefore makes sense to present tl

ifference schemes for a uniform point distribution $t_n = n\Delta t$, where Δt is the onstant spacing between the mesh points, also referred to as the *time step*. The esulting formulas look simpler and are perhaps more well known.

Summary of schemes for constant time step.

$$u^{n+1} = (1 - a\Delta t)u^n \quad \text{Forward Euler} \tag{20}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \quad \text{Backward Euler} \tag{21}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \quad \text{Crank-Nicolson} \tag{22}$$

$$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t}u^n \quad \text{The } \theta - \text{rule} \tag{23}$$

Not surprisingly, we present these three alternative schemes because they ave different pros and cons, both for the simple ODE in question (which can asily be solved as accurately as desired), and for more advanced differential quation problems.

Test the understanding.

At this point it can be good training to apply the explained finite difference discretization techniques to a slightly different equation. Exercise 10 is therefore highly recommended to check that the key concepts are understood.

.7 Compact operator notation for finite differences

inite difference formulas can be tedious to write and read, especially for differenal equations with many terms and many derivatives. To save space and help the eader of the scheme to quickly see the nature of the difference approximations, e introduce a compact notation. A forward difference approximation is denoted y the D_t^+ operator:

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \approx \frac{d}{dt}u(t_n)\,. \tag{24}$$

he notation consists of an operator that approximates differentiation with espect to an independent variable, here t. The operator is built of the symbol D, ith the variable as subscript and a superscript denoting the type of difference. he superscript $^+$ indicates a forward difference. We place square brackets round the operator and the function it operates on and specify the mesh point, here the operator is acting, by a superscript.

The corresponding operator notation for a centered difference and a ba difference reads

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt}u(t_n),$$

and

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt}u(t_n)\,.$$

Note that the superscript $^-$ denotes the backward difference, while no sup implies a central difference.

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n)$$

The superscript t indicates that the average is taken along the time coo The common average $(u^n + u^{n+1})/2$ can now be expressed as $[\overline{u}^t]^{n+\frac{1}{2}}$. also spatial coordinates enter the problem, we need the explicit specific the coordinate after the bar.)

The Backward Euler finite difference approximation to $u' = -au$ written as follows utilizing the compact notation:

$$[D_t^- u]^n = -au^n\,.$$

In difference equations we often place the square brackets around th equation, to indicate at which mesh point the equation applies, since ea is supposed to be approximated at the same point:

$$[D_t^- u = -au]^n\,.$$

The Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n,$$

while the Crank-Nicolson scheme is written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}\,.$$

Question.

Apply (25) and (27) and write out the expressions to see that (30) is ir the Crank-Nicolson scheme.

The θ-rule can be specified by

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta},$$

if we define a new time difference

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n}, \tag{32}$$

nd a *weighted averaging operator*

$$[\overline{u}^{t,\theta}]^{n+\theta} = (1-\theta)u^n + \theta u^{n+1} \approx u(t_{n+\theta}), \tag{33}$$

here $\theta \in [0,1]$. Note that for $\theta = \frac{1}{2}$ we recover the standard centered difference nd the standard arithmetic mean. The idea in (31) is to sample the equation t $t_{n+\theta}$, use a skew difference at that point $[\bar{D}_t u]^{n+\theta}$, and a skew mean value. n alternative notation is

$$[D_t u]^{n+\frac{1}{2}} = \theta[-au]^{n+1} + (1-\theta)[-au]^n \,.$$

Looking at the various examples above and comparing them with the under- ing differential equations, we see immediately which difference approximations nat have been used and at which point they apply. Therefore, the compact otation effectively communicates the reasoning behind turning a differential quation into a difference equation.

Implementation

Goal.

We want make a computer program for solving

$$u'(t) = -au(t), \quad t \in (0,T], \quad u(0) = I,$$

by finite difference methods. The program should also display the numerical solution as a curve on the screen, preferably together with the exact solution.

All programs referred to in this section are found in the `src/decay`[1] directory ve use the classical Unix term *directory* for what many others nowadays call *older*).

Mathematical problem. We want to explore the Forward Euler scheme, the ackward Euler, and the Crank-Nicolson schemes applied to our model problem. rom an implementational point of view, it is advantageous to implement the -rule

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

nce it can generate the three other schemes by various of choices of θ: $\theta = 0$ for orward Euler, $\theta = 1$ for Backward Euler, and $\theta = 1/2$ for Crank-Nicolson. Given , $u^0 = I$, T, and Δt, our task is to use the θ-rule to compute $u^1, u^2, \ldots, u^{N_t}$, here $t_{N_t} = N_t\Delta t$, and N_t the closest integer to $T/\Delta t$.

[1] `http://tinyurl.com/jvzzcfn/decay`

Computer Language: Python. Any programming language can be generate the u^{n+1} values from the formula above. However, in this do we shall mainly make use of Python of several reasons:

- Python has a very clean, readable syntax (often known as "exe pseudo-code").
- Python code is very similar to MATLAB code (and MATLAI particularly widespread use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to, but much simpler to work with and results reliable code than C++.
- Python has a rich set of modules for scientific computing, and its po in scientific computing is rapidly growing.
- Python was made for being combined with compiled languages (C Fortran) to reuse existing numerical software and to reach high c tional performance of new implementations.
- Python has extensive support for administrative task needed whe large-scale computational investigations.
- Python has extensive support for graphics (visualization, user in web applications).
- FEniCS, a very powerful tool for solving PDEs by the finite element is most human-efficient to operate from Python.

Learning Python is easy. Many newcomers to the language will probab enough from the forthcoming examples to perform their own computer ments. The examples start with simple Python code and gradually mak more powerful constructs as we proceed. As long as it is not inconven the problem at hand, our Python code is made as close as possible to M code for easy transition between the two languages.

Readers who feel the Python examples are too hard to follow will p benefit from reading a tutorial, e.g.,

- The Official Python Tutorial[2]
- Python Tutorial on tutorialspoint.com[3]
- Interactive Python tutorial site[4]
- A Beginner's Python Tutorial[5]

The author also has a comprehensive book [4] that teaches scientific progr with Python from the ground up.

[2] `http://docs.python.org/2/tutorial/`
[3] `http://www.tutorialspoint.com/python/`
[4] `http://www.learnpython.org/`
[5] `http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial`

.1 Making a solver function

We choose to have an array `u` for storing the u^n values, $n = 0, 1, \ldots, N_t$. The algorithmic steps are

1. initialize u^0
2. for $t = t_n$, $n = 1, 2, \ldots, N_t$: compute u_n using the θ-rule formula

Function for computing the numerical solution. The following Python function takes the input data of the problem (I, a, T, Δt, θ) as arguments and returns two arrays with the solution $u^0, \ldots, u^{N_t}$ and the mesh points $t_0, \ldots, t_{N_t}$, respectively:

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    Nt = int(T/dt)            # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

The `numpy` library contains a lot of functions for array computing. Most of the function names are similar to what is found in the alternative scientific computing language MATLAB. Here we make use of

- `zeros(Nt+1)` for creating an array of a size `Nt+1` and initializing the elements to zero
- `linspace(0, T, Nt+1)` for creating an array with `Nt+1` coordinates uniformly distributed between `0` and `T`

The `for` loop deserves a comment, especially for newcomers to Python. The construction `range(0, Nt, s)` generates all integers from `0` to `Nt` in steps of `s`, *but not including* `Nt`. Omitting `s` means `s=1`. For example, `range(0, 6, 3)` gives `0` and `3`, while `range(0, Nt)` generates `0`, `1`, ..., `Nt-1`. Our loop implies the following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is what we want since `u` has length `Nt+1`. The first index in Python arrays or lists is *always* 0 and the last is then `len(u)-1`. The length of an array `u` is obtained by `len(u)` or `u.size`.

To compute with the `solver` function, we need to *call* it. Here is a sample call:


```
u, t = solver(I=1, a=2, T=8, dt=0.8, theta=1)
```

Integer division. The shown implementation of the `solver` may face p and wrong results if `T`, `a`, `dt`, and `theta` are given as integers, see Exer and **??**. The problem is related to *integer division* in Python (as well as in C, C++, and many other computer languages): `1/2` becomes `0`, while `1/2.0`, or `1.0/2.0` all become `0.5`. It is enough that at least the nu or the denominator is a real number (i.e., a `float` object) to ensure mathematical division. Inserting a conversion `dt = float(dt)` guarant `dt` is `float` and avoids problems in Exercise **??**.

Another problem with computing $N_t = T/\Delta t$ is that we should roun the nearest integer. With `Nt = int(T/dt)` the `int` operation picks the integer smaller than `T/dt`. Correct mathematical rounding as known fro is obtained by

```
Nt = int(round(T/dt))
```

The complete version of our improved, safer `solver` function then beco

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Doc strings. Right below the header line in the `solver` function th Python string enclosed in triple double quotes `"""`. The purpose of thi object is to document what the function does and what the arguments this case the necessary documentation do not span more than one line, b triple double quoted strings the text may span several lines:

```
def solver(I, a, T, dt, theta):
    """
    Solve

        u'(t) = -a*u(t),

    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.
```

```
    theta=1 corresponds to the Backward Euler scheme, theta=0
    to the Forward Euler scheme, and theta=0.5 to the Crank-
    Nicolson method.
    """
    ...
```

uch documentation strings appearing right after the header of a function re called *doc strings*. There are tools that can automatically produce nicely ormatted documentation by extracting the definition of functions and the ontents of doc strings.

It is strongly recommended to equip any function whose purpose is not bvious with a doc string. Nevertheless, the forthcoming text deviates from this ule if the function is explained in the text.

ormatting of numbers. Having computed the discrete solution `u`, it is atural to look at the numbers:

```
# Write out a table of t and u values:
for i in range(len(t)):
    print t[i], u[i]
```

his compact `print` statement gives unfortunately quite ugly output because the and `u` values are not aligned in nicely formatted columns. To fix this problem, e recommend to use the *printf format*, supported most programming languages nherited from C. Another choice is Python's recent *format string syntax*.

Writing `t[i]` and `u[i]` in two nicely formatted columns is done like this ith the printf format:

```
print 't=%6.3f u=%g' % (t[i], u[i])
```

he percentage signs signify "slots" in the text where the variables listed at the nd of the statement are inserted. For each "slot" one must specify a format for ow the variable is going to appear in the string: `s` for pure text, `d` for an integer, for a real number written as compactly as possible, `9.3E` for scientific notation ith three decimals in a field of width 9 characters (e.g., `-1.351E-2`), or `.2f` for tandard decimal notation with two decimals formatted with minimum width. he printf syntax provides a quick way of formatting tabular output of numbers ith full control of the layout.

The alternative *format string syntax* looks like

```
print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

s seen, this format allows logical names in the "slots" where `t[i]` and `u[i]` are o be inserted. The "slots" are surrounded by curly braces, and the logical name followed by a colon and then the printf-like specification of how to format real umbers, integers, or strings.

Running the program. The function and main program shown abo be placed in a file, say with name `decay_v1.py`[6] (`v1` stands for "version shall make numerous different versions of this program). Make sure you w code with a suitable text editor (Gedit, Emacs, Vim, Notepad++, or s The program is run by executing the file this way:

```
Terminal> python decay_v1.py
```

The text `Terminal>` just indicates a prompt in a Unix/Linux or DOS t window. After this prompt, which will look different in your terminal depending on the terminal application and how it is set up, comma `python decay_v1.py` can be issued. These commands are interpreted operating system.

We strongly recommend to run Python programs within the IPyth First start IPython by typing `ipython` in the terminal window. Ins IPython shell, our program `decay_v1.py` is run by the command `run dec`

```
Terminal> ipython

In [1]: run decay_v1.py
t= 0.000 u=1
t= 0.800 u=0.384615
t= 1.600 u=0.147929
t= 2.400 u=0.0568958
t= 3.200 u=0.021883
t= 4.000 u=0.00841653
t= 4.800 u=0.00323713
t= 5.600 u=0.00124505
t= 6.400 u=0.000478865
t= 7.200 u=0.000184179
t= 8.000 u=7.0838e-05

In [2]:
```

The advantage of running programs in IPython are many: previous co are easily recalled with the up arrow, `%pdb` turns on debugging so that v can be examined if the program aborts due to an exception, output of co are stored in variables, programs and statements can be profiled, any o system command can be executed, modules can be loaded automatic other customizations can be performed when starting IPython – to m few of the most useful features.

Although running programs in IPython is strongly recommende execution examples in the forthcoming text use the standard Python sh prompt `>>>` and run programs through a typesetting like

[6] `http://tinyurl.com/jvzzcfn/decay/decay_v1.py`

```
erminal> python programname
```

he reason is that such typesetting makes the text more compact in the vertical irection than showing sessions with IPython syntax.

.2 Verifying the implementation

; is easy to make mistakes while deriving and implementing numerical algothms, so we should never believe in the printed u values before they have been ıoroughly verified. The most obvious idea is to compare the computed solution ith the exact solution, when that exists, but there will always be a discrepancy etween these two solutions because of the numerical approximations. The ıallenging question is whether we have the mathematically correct discrepancy r if we have another, maybe small, discrepancy due to both an approximation :ror and an error in the implementation.

The purpose of *verifying* a program is to bring evidence for the property ıat there are no errors in the implementation. To avoid mixing unavoidable pproximation errors and undesired implementation errors, we should try to ıake tests where we have some exact computation of the discrete solution or at ast parts of it. Examples will show how this can be done.

unning a few algorithmic steps by hand. The simplest approach to roduce a correct reference for the discrete solution u of finite difference equations to compute a few steps of the algorithm by hand. Then we can compare the and calculations with numbers produced by the program.

A straightforward approach is to use a calculator and compute u^1, u^2, and ^{3}. With $I = 0.1$, $\theta = 0.8$, and $\Delta t = 0.8$ we get

$$A \equiv \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t} = 0.298245614035$$

$$\begin{aligned} u^1 &= AI = 0.0298245614035, \\ u^2 &= Au^1 = 0.00889504462912, \\ u^3 &= Au^2 = 0.00265290804728 \end{aligned}$$

Comparison of these manual calculations with the result of the `solver` ınction is carried out in the function

```
lef test_solver_three_steps():
    """Compare three steps with known manual computations."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    u_by_hand = array([I,
                       0.0298245614035,
                       0.00889504462912,
                       0.00265290804728])
```



```
    Nt = 3  # number of time steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)

    tol = 1E-15  # tolerance for comparing floats
    diff = abs(u - u_by_hand).max()
    success = diff <= tol
    assert success
```

The `test_solver_three_steps` function follows widely used convent *unit testing*. By following such conventions we can at a later stage easily a big test suite for our software. The conventions are three-fold:

- The test function starts with `test_` and takes no arguments.
- The test ends up in a boolean expression that is `True` if the test and `False` if it failed.
- The function runs `assert` on the boolean expression, resulting in ı abortion (due to an `AssertionError` exception) if the test failed.

The main program, where we call the `solver` function and print `u`, is ı in a separate function `main`:

```
def main():
    u, t = solver(I=1, a=2, T=8, dt=0.8, theta=1)
    # Write out a table of t and u values:
    for i in range(len(t)):
        print 't=%6.3f u=%g' % (t[i], u[i])
        # or print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

The main program in the file may first run the verification test prior on with the real simulation (`main()`):

```
test_solver_three_steps()
main()
```

Since the verification test is always done, future errors introduced acci in the program have a good chance of being detected.

The complete program including the verification above is found in `decay_verf.py`[7].

2.3 Computing the numerical error as a mesh funct

Now that we have some evidence for a correct implementation, we a position to compare the computed u^n values in the `u` array with the values at the mesh points, in order to study the error in the numerical s

Let us first make a function for the analytical solution $u_{\mathrm{e}}(t) = Ie^{-a}$ model problem:

[7] `http://tinyurl.com/jvzzcfn/decay/decay_verf.py`

```
def exact_solution(t, I, a):
    return I*exp(-a*t)
```

A natural way to compare the exact and discrete solutions is to calculate their difference as a mesh function:

$$e^n = u_{\mathrm{e}}(t_n) - u^n, \quad n = 0, 1, \ldots, N_t\,. \tag{34}$$

We may view $u_{\mathrm{e}}^n = u_{\mathrm{e}}(t_n)$ as the representation of $u_{\mathrm{e}}(t)$ as a mesh function rather than a continuous function defined for all $t \in [0, T]$ (u_{e}^n is often called the *representative* of u_{e} on the mesh). Then, $e^n = u_{\mathrm{e}}^n - u^n$ is clearly the difference of two mesh functions. This interpretation of e^n is natural when programming.

The error mesh function e^n can be computed by

```
u, t = solver(I, a, T, dt, theta)  # Numerical sol.
u_e = exact_solution(t, I, a)      # Representative of exact sol.
e = u_e - u
```

Note that the mesh functions `u` and `u_e` are represented by arrays and associated with the points in the array `t`.

Array arithmetics.

The last statements

```
u_e = exact_solution(t, I, a)
e = u_e - u
```

are primary examples of array arithmetics: `t` is an array of mesh points that we pass to `exact_solution`. This function evaluates `-a*t`, which is a scalar times an array, meaning that the scalar is multiplied with each array element. The result is an array, let us call it `tmp1`. Then `exp(tmp1)` means applying the exponential function to each element in `tmp`, resulting an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` refers to this array returned from `exact_solution`. The expression `u_e - u` is the difference between two arrays, resulting in a new array referred to by `e`.

.4 Computing the norm of the numerical error

Instead of working with the error e^n on the entire mesh, we often want one number expressing the size of the error. This is obtained by taking the norm of the error function.

Let us first define norms of a function $f(t)$ defined for all $t \in [0, T]$. Three common norms are

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt\right)^{1/2},$$

$$||f||_{L^1} = \int_0^T |f(t)| dt,$$

$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,.$$

The L^2 norm (35) ("L-two norm") has nice mathematical properties an most popular norm. It is a generalization of the well-known Eucledia of vectors to functions. The L^∞ is also called the max norm or the su norm. In fact, there is a whole family of norms,

$$||f||_{L^p} = \left(\int_0^T f(t)^p dt\right)^{1/p},$$

with p real. In particular, $p = 1$ corresponds to the L^1 norm above whil is the L^∞ norm.

Numerical computations involving mesh functions need correspondin Given a set of function values, f^n, and some associated mesh point numerical integration rule can be used to calculate the L^2 and L^1 norms above. Imagining that the mesh function is extended to vary linearly b the mesh points, the Trapezoidal rule is in fact an exact integration possible modification of the L^2 norm for a mesh function f^n on a unifor with spacing Δt is therefore the well-known Trapezoidal integration for

$$||f^n|| = \left(\Delta t\left(\frac{1}{2}(f^0)^2 + \frac{1}{2}(f^{N_t})^2 + \sum_{n=1}^{N_t-1}(f^n)^2\right)\right)^{1/2}$$

A common approximation of this expression, motivated by the conven having a simpler formula, is

$$||f^n||_{\ell^2} = \left(\Delta t\sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}.$$

This is called the discrete L^2 norm and denoted by ℓ^2. The error i compared with the Trapezoidal integration formula is $\Delta t((f^0)^2 + (f$ which means perturbed weights at the end points of the mesh function, error goes to zero as $\Delta t \to 0$. As long as we are consistent and stick to c of integration rule for the norm of a mesh function, the details and acc this rule is not of concern.

The three discrete norms for a mesh function f^n, corresponding to L^1, and L^∞ norms of $f(t)$ defined above, are defined by

$$||f^n||_{\ell^2}\left(\Delta t\sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}, \tag{39}$$

$$||f^n||_{\ell^1}\Delta t\sum_{n=0}^{N_t}|f^n| \tag{40}$$

$$||f^n||_{\ell^\infty}\max_{0\leq n\leq N_t}|f^n|\,. \tag{41}$$

Note that the L^2, L^1, ℓ^2, and ℓ^1 norms depend on the length of the interval f interest (think of $f=1$, then the norms are proportional to $\sqrt{T}$ or T). In ome applications it is convenient to think of a mesh function as just a vector of unction values and neglect the information of the mesh points. Then we can eplace Δt by T/N_t and drop T. Moreover, it is convenient to divide by the otal length of the vector, N_t+1, instead of N_t. This reasoning gives rise to the *ector norms* for a vector $f=(f_0,\ldots,f_N)$:

$$||f||_2=\left(\frac{1}{N+1}\sum_{n=0}^{N}(f_n)^2\right)^{1/2}, \tag{42}$$

$$||f||_1=\frac{1}{N+1}\sum_{n=0}^{N}|f_n| \tag{43}$$

$$||f||_{\ell^\infty}=\max_{0\leq n\leq N}|f_n|\,. \tag{44}$$

ere we have used the common vector component notation with subscripts (f_n) nd N as length. We will mostly work with mesh functions and use the discrete ² norm (39) or the max norm ℓ^∞ (41), but the corresponding vector norms 42)-(44) are also much used in numerical computations, so it is important to now the different norms and the relations between them.

A single number that expresses the size of the numerical error will be taken s $||e^n||_{\ell^2}$ and called E:

$$E=\sqrt{\Delta t\sum_{n=0}^{N_t}(e^n)^2} \tag{45}$$

he corresponding Python code, using array arithmetics, reads

```
E = sqrt(dt*sum(e**2))
```

he `sum` function comes from `numpy` and computes the sum of the elements of n array. Also the `sqrt` function is from `numpy` and computes the square root of ach element in the array argument.

Scalar computing. Instead of doing array computing `sqrt(dt*sum` we can compute with one element at a time:

```
m = len(u)     # length of u array (alt: u.size)
u_e = zeros(m)
t = 0
for i in range(m):
    u_e[i] = exact_solution(t, a, I)
    t = t + dt
e = zeros(m)
for i in range(m):
    e[i] = u_e[i] - u[i]
s = 0  # summation variable
for i in range(m):
    s = s + e[i]**2
error = sqrt(dt*s)
```

Such element-wise computing, often called *scalar* computing, takes mo is less readable, and runs much slower than what we can achieve wit computing.

2.5 Plotting solutions

Having the `t` and `u` arrays, the approximate solution `u` is visualized intuitive command `plot(t, u)`:

```
from matplotlib.pyplot import *
plot(t, u)
show()
```

Plotting multiple curves. It will be illustrative to also plot $u_e(t)$ f parison. Doing a `plot(t, u_e)` is not exactly what we want: the `plot` draws straight lines between the discrete points `(t[n], u_e[n])` wh varies as an exponential function between the mesh points. The techn showing the "exact" variation of $u_e(t)$ between the mesh points is to intr very fine mesh for $u_e(t)$:

```
t_e = linspace(0, T, 1001)      # fine mesh
u_e = exact_solution(t_e, I, a)
plot(t_e, u_e, 'b-')            # blue line for u_e
plot(t,   u,   'r--o')          # red dashes w/circles
```

With more than one curve in the plot we need to associate eac with a legend. We also want appropriate names on the axis, a title, a containing the plot as an image for inclusion in reports. The Matplotlib (`matplotlib.pyplot`) contains functions for this purpose. The name functions are similar to the plotting functions known from MATLAB. A c plot session then becomes

```
from matplotlib.pyplot import *

figure()                          # create new plot
t_e = linspace(0, T, 1001)        # fine mesh for u_e
u_e = exact_solution(t_e, I, a)
plot(t,   u,   'r--o')            # red dashes w/circles
plot(t_e, u_e, 'b-')              # blue line for exact sol.
legend(['numerical', 'exact'])
xlabel('t')
ylabel('u')
title('theta=%g, dt=%g' % (theta, dt))
savefig('%s_%g.png' % (theta, dt))
show()
```

Note that `savefig` here creates a PNG file whose name reflects the values of θ and Δt so that we can easily distinguish files from different runs with θ and Δt.

A bit more sophisticated and easy-to-read filename can be generated by mapping the θ value to acronyms for the three common schemes: FE (Forward Euler, $\theta = 0$), BE (Backward Euler, $\theta = 1$), CN (Crank-Nicolson, $\theta = 0.5$). A Python dictionary is ideal for such a mapping from numbers to strings:

```
theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
savefig('%s_%g.png' % (theta2name[theta], dt))
```

Experiments with computing and plotting. Let us wrap up the computation of the error measure and all the plotting statements in a function `explore`. This function can be called for various θ and Δt values to see how the error varies with the method and the mesh resolution:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).
    """
    u, t = solver(I, a, T, dt, theta)    # Numerical solution
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                         # create new plot
        t_e = linspace(0, T, 1001)       # fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'r--o')           # red dashes w/circles
        plot(t_e, u_e, 'b-')             # blue line for exact sol.
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        savefig('%s_%g.eps' % (theta2name[theta], dt))
        show()
    return E
```


The `figure()` call is key here: without it, a new `plot` command w the new pair of curves in the same plot window, while we want the c pairs to appear in separate windows and files. Calling `figure()` ensure

The `explore` function stores the plot in three different image file f PNG, PDF, and EPS (Encapsulated PostScript). The PNG format is a being included in HTML files, the PDF format in pdfLaTeX documents, EPS format in LaTeX documents. Frequently used viewers for these ima on Unix systems are `gv` (comes with Ghostscript) for the PDF and EPS and `display` (from the ImageMagick) suite for PNG files:

```
Terminal> gv BE_0.5.pdf
Terminal> gv BE_0.5.eps
Terminal> display BE_0.5.png
```

The complete code containing the functions above resides in the file `plot_mpl.py`[8]. Running this program results in

```
Terminal> python decay_plot_mpl.py
0.0   0.40:    2.105E-01
0.0   0.04:    1.449E-02
0.5   0.40:    3.362E-02
0.5   0.04:    1.887E-04
1.0   0.40:    1.030E-01
1.0   0.04:    1.382E-02
```

We observe that reducing Δt by a factor of 10 increases the accuracy three methods (θ values). We also see that the combination of $\theta = 0.$ small time step $\Delta t = 0.04$ gives a much more accurate solution, and tha and $\theta = 1$ with $\Delta t = 0.4$ result in the least accurate solutions.

Figure 6 demonstrates that the numerical solution for $\Delta t = 0.4$ cle below the exact curve, but that the accuracy improves considerably by r the time step by a factor of 10.

Combining plot files. Mounting two PNG files, as done in the figure, done by the `montage`[9] program from the ImageMagick suite:

```
Terminal> montage -background white -geometry 100% -tile 2x1 \
          FE_0.4.png FE_0.04.png FE1.png
Terminal> convert -trim FE1.png FE1.png
```

The `-geometry` argument is used to specify the size of the image, and preserve the individual sizes of the images. The `-tile HxV` option sp

[8] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py
[9] http://www.imagemagick.org/script/montage.php

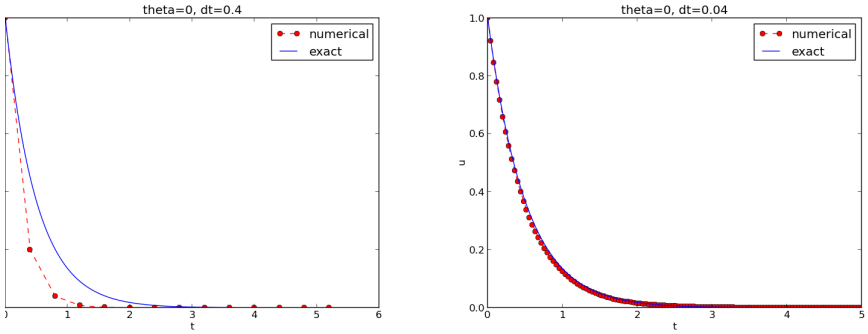


Figure 6: The Forward Euler scheme for two values of the time step.

nages in the horizontal direction and `V` images in the vertical direction. A series f image files to be combined are then listed, with the name of the resulting ombined image, here `FE1.png` at the end. The `convert -trim` command emoves surrounding white areas in the figure (an operation usually known as *opping* in image manipulation programs).

For LaTeX reports it is not recommended to use `montage` and PNG files as the esult has too low resolution. Instead, plots should be made in the PDF format nd combined using the `pdftk`, `pdfnup`, and `pdfcrop` tools (on Linux/Unix):

```
erminal> pdftk FE_0.4.png FE_0.04.png output tmp.pdf
erminal> pdfnup --nup 2x1 tmp.pdf      # output in tmp-nup.pdf
erminal> pdfcrop tmp-nup.pdf FE1.png   # output in FE1.png
```

ere, `pdftk` combines images into a multi-page PDF file, `pdfnup` combines the nages in individual pages to a table of images (pages), and `pdfcrop` removes hite margins in the resulting combined image file.

The behavior of the two other schemes is shown in Figures 7 and 8. Crank-icolson is obviously the most accurate scheme from this visual point of view.

lotting with SciTools. The SciTools package[10] provides a unified plotting nterface, called Easyviz, to many different plotting packages, including Mat-lotlib, Gnuplot, Grace, MATLAB, VTK, OpenDX, and VisIt. The syntax is ery similar to that of Matplotlib and MATLAB. In fact, the plotting commands nown above look the same in SciTool's Easyviz interface, apart from the import tatement, which reads

[10] `http://code.google.com/p/scitools`

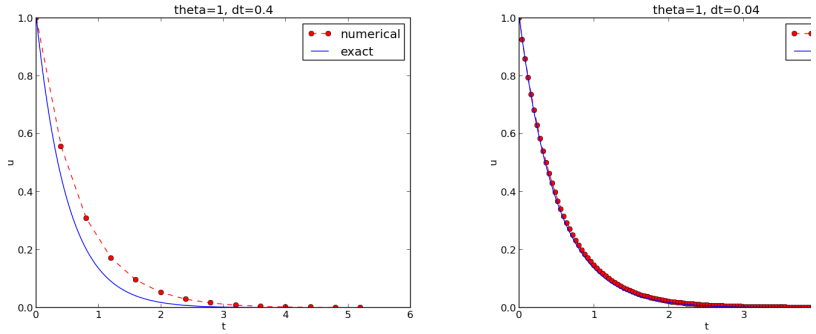


Figure 7: The Backward Euler scheme for two values of the time st

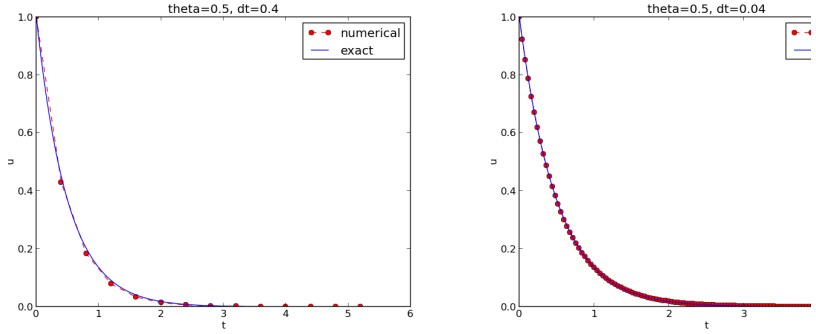


Figure 8: The Crank-Nicolson scheme for two values of the time st

```
from scitools.std import *
```

This statement performs a `from numpy import *` as well as an impor most common pieces of the Easyviz (`scitools.easyviz`) package, alo some additional numerical functionality.

With Easyviz one can merge several plotting commands into a sir using keyword arguments:

```
plot(t,   u,   'r--o',          # red dashes w/circles
     t_e, u_e, 'b-',            # blue line for exact sol.
     legend=['numerical', 'exact'],
     xlabel='t',
     ylabel='u',
     title='theta=%g, dt=%g' % (theta, dt),
     savefig='%s_%g.png' % (theta2name[theta], dt),
     show=True)
```

he `decay_plot_st.py`[11] file contains such a demo.

By default, Easyviz employs Matplotlib for plotting, but Gnuplot[12] and Grace[13] are viable alternatives:

```
erminal> python decay_plot_st.py --SCITOOLS_easyviz_backend gnuplot
erminal> python decay_plot_st.py --SCITOOLS_easyviz_backend grace
```

he backend used for creating plots (and numerous other options) can be ermanently set in SciTool's configuration file.

All the Gnuplot windows are launched without any need to kill one before ıe next one pops up (as is the case with Matplotlib) and one can press the key ' anywhere in a plot window to kill it. Another advantage of Gnuplot is the utomatic choice of sensible and distinguishable line types in black-and-white DF and PostScript files.

Regarding functionality for annotating plots with title, labels on the axis, gends, etc., we refer to the documentation of Matplotlib and SciTools for more etailed information on the syntax. The hope is that the programming syntax xplained so far suffices for understanding the code and learning more from a ombination of the forthcoming examples and other resources such as books and eb pages.

Test the understanding.

Exercise 11 asks you to implement a solver for a problem that is slightly different from the one above. You may use the `solver` and `explore` functions explained above as a starting point. Apply the new solver to Exercise 12.

.6 Memory-saving implementation

he computer memory requirements of our implementations so far consists ıainly of the `u` and `t` arrays, both of length $N_t + 1$, plus some other temporary rrays that Python needs for intermediate results if we do array arithmetics in ur program (e.g., `I*exp(-a*t)` needs to store `a*t` before `-` can be applied to it nd then `exp`). Regardless of how we implement simple ODE problems, storage equirements are very modest and put not restriction on how we choose our data tructures and algorithms. Nevertheless, when the methods for ODEs used here re applied to three-dimensional partial differential equation (PDE) problems, ıemory storage requirements suddenly become a challenging issue.

[11] `http://tinyurl.com/jvzzcfn/decay/decay_plot_st.py`
[12] `http://www.gnuplot.info/`
[13] `http://plasma-gate.weizmann.ac.il/Grace/`

The PDE counterpart to our model problem $u' = -a$ is a diffusion e $u_t = a\nabla^2 u$ posed on a space-time domain. The discrete representatior domain may in 3D be a spatial mesh of M^3 points and a time mes points. A typical desired value for M is 100 in many applications, or ev Storing all the computed u values, like we have done in the program demands storage of some arrays of size $M^3 N_t$, giving a factor of M storage demands compared to our ODE programs. Each real numbe array for u requires 8 bytes (b) of storage. With $M = 100$ and N_t there is a storage demand of $(10^3)^3 \cdot 1000 \cdot 8 = 8$ Gb for the solutio Fortunately, we can usually get rid of the N_t factor, resulting in 8 Mb of Below we explain how this is done, and the technique is almost always in implementations of PDE problems.

Let us critically evaluate how much we really need to store in the cor memory in our implementation of the θ method. To compute a new u^{n+} need is u^n. This implies that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do r to be stored in an array, although this is convenient for plotting and data in the program. Instead of the `u` array we can work with two variables numbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorithm, resp At each time level, we update `u` from `u_1` and then set `u_1 = u` so t computed u^{n+1} value becomes the "previous" value u^n at the next tir The downside is that we cannot plot the solution after the simulation since only the last two numbers are available. The remedy is to store co values in a file and use the file for visualizing the solution later.

We have implemented this memory saving idea in the file `decay_m py`[14], which is a slight modification of `decay_plot_mpl.py`[15] program.

The following function demonstrates how we work with the two mos values of the unknown:

```
def solver_memsave(I, a, T, dt, theta, filename='sol.dat'):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.
    Minimum use of memory. The solution is stored in a file
    (with name filename) for later plotting.
    """
    dt = float(dt)         # avoid integer division
    Nt = int(round(T/dt))  # no of intervals

    outfile = open(filename, 'w')
    # u: time level n+1, u_1: time level n
    t = 0
    u_1 = I
    outfile.write('%.16E  %.16E\n' % (t, u_1))
    for n in range(1, Nt+1):
        u = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u_1
        u_1 = u
        t += dt
        outfile.write('%.16E  %.16E\n' % (t, u))
    outfile.close()
    return u, t
```

[14] `http://tinyurl.com/jvzzcfn/decay/decay_memsave.py`
[15] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`

his code snippet serves as a quick introduction to file writing in Python. Reading
ıe data in the file into arrays `t` and `u` are done by the function

```
lef read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = [];  t = []
    for line in infile:
        words = line.split()
        if len(words) != 2:
            print 'Found more than two numbers on a line!', words
            sys.exit(1)  # abort
        t.append(float(words[0]))
        u.append(float(words[1]))
    return np.array(t), np.array(u)
```

This type of file with numbers in rows and columns is very common, and `numpy`
as a function `loadtxt` which loads such tabular data into a two-dimensional
rray, say with name `data`. The number in row `i` and column `j` is then `data[i,j]`.
he whole column number `j` can be extracted by `data[:,j]`. A version of
ead_file using `np.loadtxt` reads

```
lef read_file_numpy(filename='sol.dat'):
    data = np.loadtxt(filename)
    t = data[:,0]
    u = data[:,1]
    return t, u
```

The present counterpart to the `explore` function from `decay_plot_mpl.py`[16]
ıust run `solver_memsave` and then load data from file before we can compute
ıe error measure and make the plot:

```
lef explore(I, a, T, dt, theta=0.5, makeplot=True):
    filename = 'u.dat'
    u, t = solver_memsave(I, a, T, dt, theta, filename)

    t, u = read_file(filename)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*np.sum(e**2))
    if makeplot:
        figure()
        ...
```

Apart from the internal implementation, where u^n values are stored in
file rather than in an array, `decay_memsave.py` file works exactly as the
ecay_plot_mpl.py file.

Analysis of finite difference equations

/e address the ODE for exponential decay,

$$u'(t) = -au(t), \quad u(0) = I, \tag{46}$$

[16] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`

where a and I are given constants. This problem is solved by the θ-ru
difference scheme, resulting in the recursive equations

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n$$

for the numerical solution u^{n+1}, which approximates the exact solution u_e
point t_{n+1}. For constant mesh spacing, which we assume here, $t_{n+1} = (n$

Discouraging numerical solutions. Choosing $I = 1$, $a = 2$, and
experiments with $\theta = 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the re
Figures 9, 10, and 11.

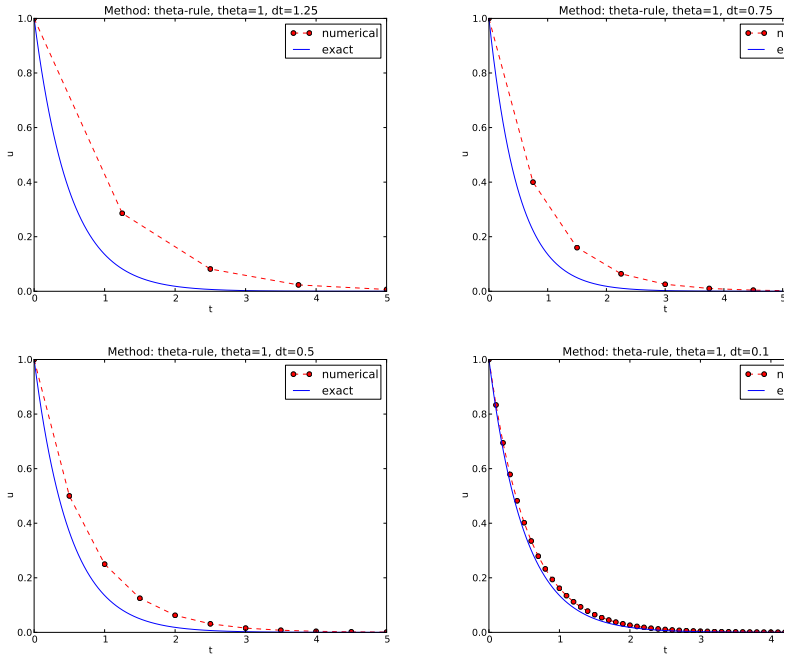


Figure 9: Backward Euler.

The characteristics of the displayed curves can be summarized as fo

- The Backward Euler scheme always gives a monotone solution, lyin
  the exact curve.
- The Crank-Nicolson scheme gives the most accurate results, but f
  1.25 the solution oscillates.

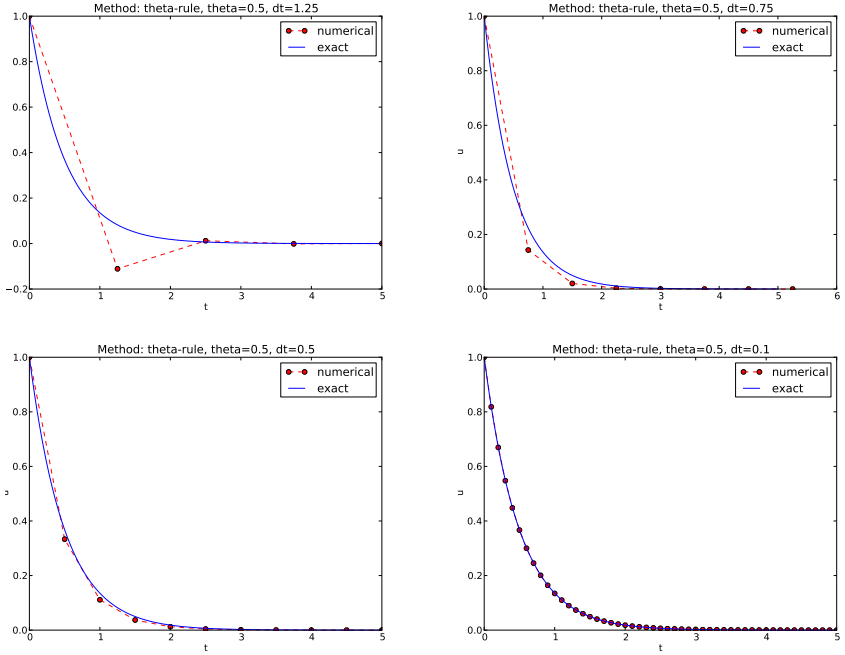


Figure 10: Crank-Nicolson.

- The Forward Euler scheme gives a growing, oscillating solution for $\Delta t = 1.25$; a decaying, oscillating solution for $\Delta t = 0.75$; a strange solution $u^n = 0$ for $n \geq 1$ when $\Delta t = 0.5$; and a solution seemingly as accurate as the one by the Backward Euler scheme for $\Delta t = 0.1$, but the curve lies below the exact solution.

ince the exact solution of our model problem is a monotone function, $u(t) = e^{-at}$, some of these qualitatively wrong results are indeed alarming!

Goal.

We ask the question

- Under what circumstances, i.e., values of the input data I, a, and Δt will the Forward Euler and Crank-Nicolson schemes result in undesired oscillatory solutions?

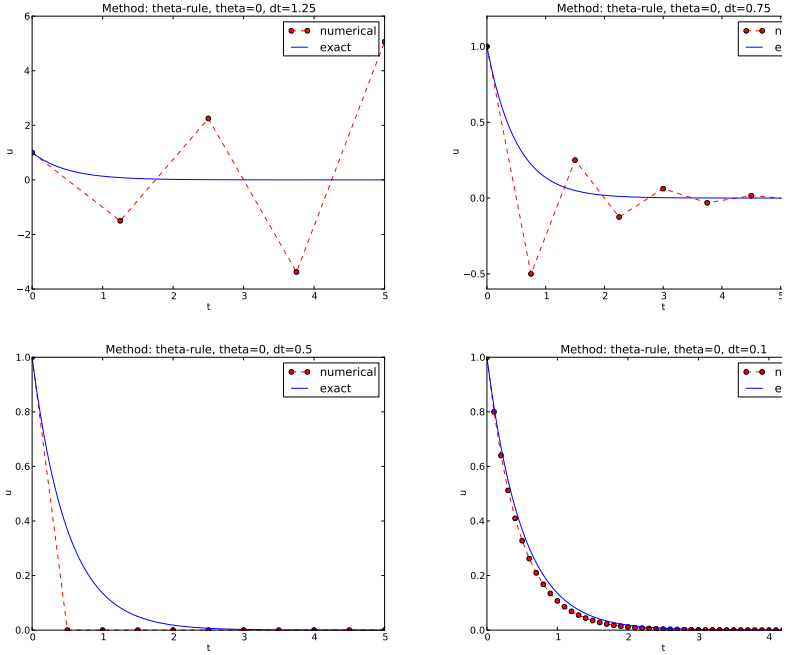


Figure 11: Forward Euler.

The question will be investigated both by numerical experiments ar precise mathematical theory. The latter will help establish general cr on Δt for avoiding non-physical oscillatory or growing solutions.

Another question to be raised is

- How does Δt impact the error in the numerical solution?

For our simple model problem we can answer this question very prec but we will also look at simplified formulas for small Δt and touch important concepts such as *convergence rate* and *the order of a sc* Other fundamental concepts mentioned are stability, consistency, convergence.

3.1 Experimental investigation of oscillatory solutic

To address the first question above, we may set up an experiment w loop over values of I, a, and Δt. For each experiment, we flag the sol oscillatory if

$$u^n > u^{n-1},$$

or some value of n, since we expect u^n to decay with n, but oscillations make u increase over a time step. We will quickly see that oscillations are independent of I, but do depend on a and Δt. Therefore, we introduce a two-dimensional function $B(a, \Delta t)$ which is 1 if oscillations occur and 0 otherwise. We can visualize B as a contour plot (lines for which $B = \text{const}$). The contour $B = 0.5$ corresponds to the borderline between oscillatory regions with $B = 1$ and monotone regions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have given P a values, $a_0, \ldots, a_{P-1}$, and Q Δt values, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, $i = 0, \ldots, P-1$, $j = 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the plane. At each point $(a_i, \Delta t_j)$, we associate the corresponding value of $B(a_i, \Delta t_j)$, denoted B_{ij}. The B_{ij} values are naturally stored in a two-dimensional array. We can thereafter create a plot of the contour line $B_{ij} = 0.5$ dividing the oscillatory and monotone regions. The file `decay_osc_regions.py`[17] `osc_regions` stands for "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ line in Figures 12 and 13. The oscillatory region is above this line.

```
from decay_mod import solver
import numpy as np
import scitools.std as st

def non_physical_behavior(I, a, T, dt, theta):
    """
    Given lists/arrays a and dt, and numbers I, dt, and theta,
    make a two-dimensional contour line B=0.5, where B=1>0.5
    means oscillatory (unstable) solution, and B=0<0.5 means
    monotone solution of u'=-au.
    """
    a = np.asarray(a); dt = np.asarray(dt)  # must be arrays
    B = np.zeros((len(a), len(dt)))         # results
    for i in range(len(a)):
        for j in range(len(dt)):
            u, t = solver(I, a[i], T, dt[j], theta)
            # Does u have the right monotone decay properties?
            correct_qualitative_behavior = True
            for n in range(1, len(u)):
                if u[n] > u[n-1]:  # Not decaying?
                    correct_qualitative_behavior = False
                    break  # Jump out of loop
            B[i,j] = float(correct_qualitative_behavior)
    a_, dt_ = st.ndgrid(a, dt)  # make mesh of a and dt values
    st.contour(a_, dt_, B, 1)
    st.grid('on')
    st.title('theta=%g' % theta)
    st.xlabel('a'); st.ylabel('dt')
    st.savefig('osc_region_theta_%s.png' % theta)
    st.savefig('osc_region_theta_%s.pdf' % theta)

non_physical_behavior(
    I=1,
    a=np.linspace(0.01, 4, 22),
    dt=np.linspace(0.01, 4, 22),
    T=6,
    theta=0.5)
```

[17] http://tinyurl.com/jvzzcfn/decay/decay_osc_regions.py

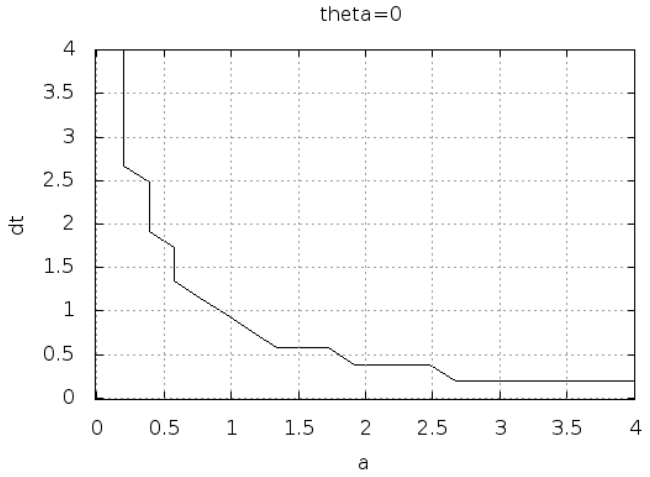


Figure 12: Forward Euler scheme: oscillatory solutions occur for point the curve.

By looking at the curves in the figures one may guess that $a\Delta t$ must than a critical limit to avoid the undesired oscillations. This limit seen about 2 for Crank-Nicolson and 1 for Forward Euler. We shall now e a precise mathematical analysis of the discrete model that can expl observations in our numerical experiments.

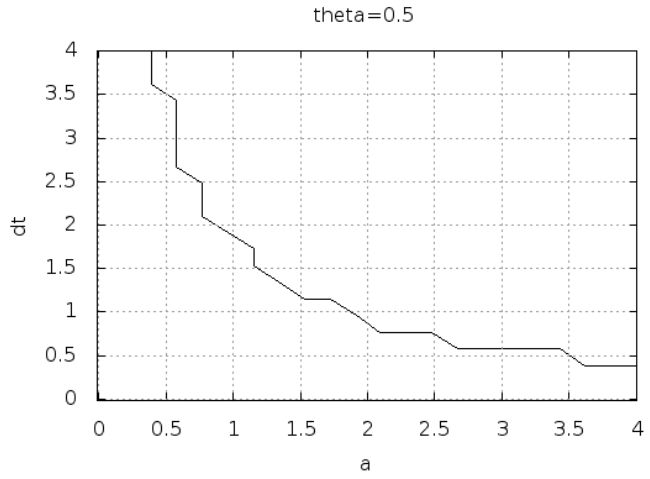
3.2 Exact numerical solution

Starting with $u^0 = I$, the simple recursion (47) can be applied repea times, with the result that

$$u^n = IA^n, \quad A = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a \Delta t}.$$

Solving difference equations.

Difference equations where all terms are linear in u^{n+1}, u^n, and m u^{n-1}, u^{n-2}, etc., are called *homogeneous, linear* difference equations their solutions are generally of the form $u^n = A^n$. Inserting this expre and dividing by A^{n+1} gives a polynomial equation in A. In the pr



igure 13: Crank-Nicolson scheme: oscillatory solutions occur for points above ıe curve.

case we get

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,.$$

This is a solution technique of wider applicability than repeated use of the recursion (47).

Regardless of the solution approach, we have obtained a formula for u^n. This ɔrmula can explain everything what we see in the figures above, but it also ives us a more general insight into accuracy and stability properties of the three :hemes.

.3 Stability

ince u^n is a factor A raised to an integer power n, we realize that $A < 0$ will ɔr odd powers imply $u^n < 0$ and for even power result in $u^n > 0$. That is, the ɔlution oscillates between the mesh points. We have oscillations due to $A < 0$ hen

$$(1-\theta)a\Delta t > 1\,. \tag{49}$$

Since $A > 0$ is a requirement for having a numerical solution with tl basic property (monotonicity) as the exact solution, we may say that A *stability criterion*. Expressed in terms of Δt the stability criterion read

$$\Delta t < \frac{1}{(1-\theta)a}\,.$$

The Backward Euler scheme is always stable since $A < 0$ is impos $\theta = 1$, while non-oscillating solutions for Forward Euler and Crank-N demand $\Delta t \leq 1/a$ and $\Delta t \leq 2/a$, respectively. The relation between Δ look reasonable: a larger a means faster decay and hence a need for smal steps.

Looking at Figure 11, we see that with $a\Delta t = 2 \cdot 1.25 = 2.5$, $A = -$ the solution $u^n = (-1.5)^n$ oscillates *and* grows. With $a\Delta t = 2 \cdot 0.7$ $A = -0.5$, $u^n = (-0.5)^n$ decays but oscillates. The peculiar case Δ where the Forward Euler scheme produces a solution that is stuck on th corresponds to $A = 0$ and therefore $u^0 = I = 1$ and $u^n = 0$ for $n \geq$ decaying oscillations in the Crank-Nicolson scheme for $\Delta t = 1.25$ ar explained by the fact that $A \approx -0.11 < 0$.

The factor A is called the *amplification factor* since the solution at a n level is A times the solution at the previous time level. For a decay pro must obviously have $|A| \leq 1$, which is fulfilled for all Δt if $\theta \geq 1/2$. Arl large values of u can be generated when $|A| > 1$ and n is large enoug numerical solution is in such cases totally irrelevant to an ODE modelir processes! To avoid this situation, we must for $\theta < 1/2$ have

$$\Delta t \leq \frac{2}{(1-2\theta)a},$$

which means $\Delta t < 2/a$ for the Forward Euler scheme.

Stability properties.

We may summarize the stability investigations as follows:

1. The Forward Euler method is a *conditionally stable* scheme be it requires $\Delta t < 2/a$ for avoiding growing solutions and $\Delta t < 1/$ avoiding oscillatory solutions.

2. The Crank-Nicolson is *unconditionally stable* with respect to gr solutions, while it is conditionally stable with the criterion $\Delta t <$ for avoiding oscillatory solutions.

3. The Backward Euler method is unconditionally stable with re to growing and oscillatory solutions - any Δt will work.

Much literature on ODEs speaks about L-stable and A-stable methods. In our case A-stable methods ensures non-growing solutions, while L-stable methods also avoids oscillatory solutions.

.4 Comparing amplification factors

fter establishing how A impacts the qualitative features of the solution, we shall ow look more into how well the numerical amplification factor approximates ıe exact one. The exact solution reads $u(t) = Ie^{-at}$, which can be rewritten as

$$u_{\mathrm{e}}(t_n) = Ie^{-an\Delta t} = I(e^{-a\Delta t})^n\,. \tag{52}$$

rom this formula we see that the exact amplification factor is

$$A_{\mathrm{e}} = e^{-a\Delta t}\,. \tag{53}$$

We realize that the exact and numerical amplification factors depend on a and t through the product $a\Delta t$. Therefore, it is convenient to introduce a symbol ɔr this product, $p = a\Delta t$, and view A and A_{e} as functions of p. Figure 14 shows ıese functions. Crank-Nicolson is clearly closest to the exact amplification ıctor, but that method has the unfortunate oscillatory behavior when $p > 2$.

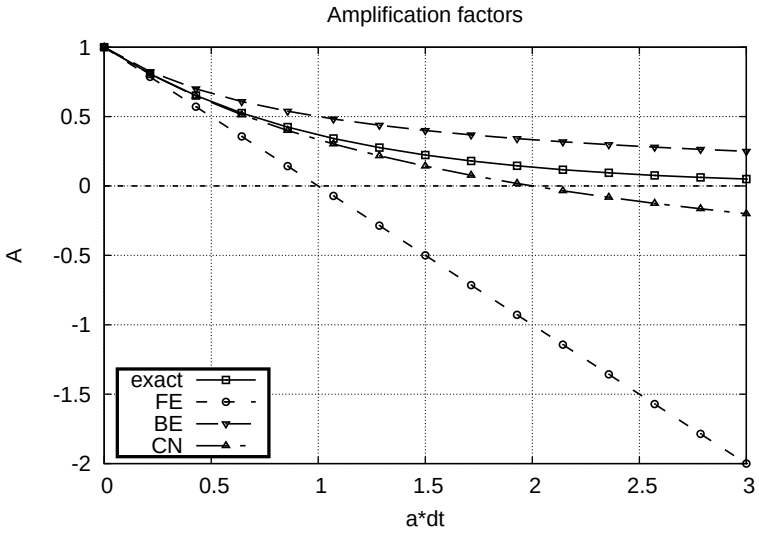


Figure 14: Comparison of amplification factors.

3.5 Series expansion of amplification factors

As an alternative to the visual understanding inherent in Figure 14, tl strong tradition in numerical analysis to establish formulas for the approx errors when the discretization parameter, here Δt, becomes small. In the case we let p be our small discretization parameter, and it makes sense to the expressions for A and A_{e} by using Taylor polynomials around $p =$ Taylor polynomials are accurate for small p and greatly simplifies the com of the analytical expressions since we then can compare polynomials, term.

Calculating the Taylor series for A_{e} is easily done by hand, but tl versions of A for $\theta = 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. No analytical computations can benefit greatly by symbolic computer algel ware. The Python package `sympy` represents a powerful computer algebra not yet as sophisticated as the famous Maple and Mathematica syste free and very easy to integrate with our numerical computations in Pyt

When using `sympy`, it is convenient to enter the interactive Pytho where we can write expressions and statements and immediately see the Here is a simple example. We strongly recommend to use `isympy` (or `i` for such interactive sessions.

Let us illustrate `sympy` with a standard Python shell syntax (`>>>` pro compute a Taylor polynomial approximation to e^{-p}:

```
>>> from sympy import *
>>> # Create p as a mathematical symbol with name 'p'
>>> p = Symbol('p')
>>> # Create a mathematical expression with p
>>> A_e = exp(-p)
>>>
>>> # Find the first 6 terms of the Taylor series of A_e
>>> A_e.series(p, 0, 6)
1 + (1/2)*p**2 - p - 1/6*p**3 - 1/120*p**5 + (1/24)*p**4 + O(p**6
```

Lines with `>>>` represent input lines and lines without this prompt rep the result of computations (note that `isympy` and `ipython` apply other p but in this text we always apply `>>>` for interactive Python computing) from the order of the powers, the computed formula is easily recognize beginning of the Taylor series for e^{-p}.

Let us define the numerical amplification factor where p and θ e formula as symbols:

```
>>> theta = Symbol('theta')
>>> A = (1-(1-theta)*p)/(1+theta*p)
```

To work with the factor for the Backward Euler scheme we can substit value 1 for `theta`:

```
>>> A.subs(theta, 1)
1/(1 + p)
```

imilarly, we can replace `theta` by 1/2 for Crank-Nicolson, preferably using an xact rational representation of 1/2 in `sympy`:

```
>>> half = Rational(1,2)
>>> A.subs(theta, half)
1/(1 + (1/2)*p)*(1 - 1/2*p)
```

The Taylor series of the amplification factor for the Crank-Nicolson scheme an be computed as

```
>>> A.subs(theta, half).series(p, 0, 4)
1 + (1/2)*p**2 - p - 1/4*p**3 + O(p**4)
```

We are now in a position to compare Taylor series:

```
>>> FE = A_e.series(p, 0, 4) - A.subs(theta, 0).series(p, 0, 4)
>>> BE = A_e.series(p, 0, 4) - A.subs(theta, 1).series(p, 0, 4)
>>> CN = A_e.series(p, 0, 4) - A.subs(theta, half).series(p, 0, 4 )
>>> FE
(1/2)*p**2 - 1/6*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (5/6)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

rom these expressions we see that the error $A - A_\mathrm{e} \sim \mathcal{O}(p^2)$ for the Forward nd Backward Euler schemes, while $A - A_\mathrm{e} \sim \mathcal{O}(p^3)$ for the Crank-Nicolson cheme. It is the *leading order term*, i.e., the term of the lowest order (polynomial egree), that is of interest, because as $p \to 0$, this term is (much) bigger than ne higher-order terms (think of $p = 0.01$: p is a hundred times larger than p^2).

Now, a is a given parameter in the problem, while Δt is what we can vary. ne therefore usually writes the error expressions in terms Δt. When then have

$$A - A_\mathrm{e} = \begin{cases} \mathcal{O}(\Delta t^2), & \text{Forward and Backward Euler,} \\ \mathcal{O}(\Delta t^3), & \text{Crank-Nicolson} \end{cases} \tag{54}$$

We say that the Crank-Nicolson scheme has an error in the amplification ictor of order Δt^3, while the two other schemes are of order Δt^2 in the same uantity. What is the significance of the order expression? If we halve Δt, ne error in amplification factor at a time level will be reduced by a factor of in the Forward and Backward Euler schemes, and by a factor of 8 in the rank-Nicolson scheme. That is, as we reduce Δt to obtain more accurate esults, the Crank-Nicolson scheme reduces the error more efficiently than the ther schemes.

3.6 The fraction of numerical and exact amplificatic tors

An alternative comparison of the schemes is to look at the ratio A/A_e error $1 - A/A_\mathrm{e}$ in this ratio:

```
>>> FE = 1 - (A.subs(theta, 0)/A_e).series(p, 0, 4)
>>> BE = 1 - (A.subs(theta, 1)/A_e).series(p, 0, 4)
>>> CN = 1 - (A.subs(theta, half)/A_e).series(p, 0, 4)
>>> FE
(1/2)*p**2 + (1/3)*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (1/3)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

The leading-order terms have the same powers as in the analysis of A -

3.7 The global error at a point

The error in the amplification factor reflects the error when progressi time level t_n to t_{n-1}. To investigate the real error at a point, know *global error*, we look at $e^n = u^n - u_\mathrm{e}(t_n)$ for some n and Taylor exp mathematical expressions as functions of $p = a\Delta t$:

```
>>> n = Symbol('n')
>>> u_e = exp(-p*n)
>>> u_n = A**n
>>> FE = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
>>> BE = u_e.series(p, 0, 4) - u_n.subs(theta, 1).series(p, 0, 4)
>>> CN = u_e.series(p, 0, 4) - u_n.subs(theta, half).series(p, 0,
>>> FE
(1/2)*n*p**2 - 1/2*n**2*p**3 + (1/3)*n*p**3 + O(p**4)
>>> BE
(1/2)*n**2*p**3 - 1/2*n*p**2 + (1/3)*n*p**3 + O(p**4)
>>> CN
(1/12)*n*p**3 + O(p**4)
```

For a fixed time t, the parameter n in these expressions increases as $p \to$ $t = n\Delta t$ = const and hence n must increase like Δt^{-1}. With n sub by $t/\Delta t$ in the leading-order error terms, these become $\frac{1}{2}na^2\Delta t^2 = \frac{1}{2}ta$ the Forward and Backward Euler scheme, and $\frac{1}{12}na^3\Delta t^3 = \frac{1}{12}ta^3\Delta t^2$ Crank-Nicolson scheme. The global error is therefore of second order (in the latter scheme and of first order for the former schemes.

When the global error $e^n \to 0$ as $\Delta t \to 0$, we say that the scheme is *co* It means that the numerical solution approaches the exact solution as t is refined, and this is a much desired property of a numerical method.

3.8 Integrated errors

It is common to study the norm of the numerical error, as explained in Section 2.4. The L^2 norm can be computed by treating e^n as a functio

ympy and performing symbolic integration. For the Forward Euler scheme we ave

```
, n, a, dt, t, T, theta = symbols('p n a dt t T 'theta')
 = (1-(1-theta)*p)/(1+theta*p)
_e = exp(-p*n)
_n = A**n
rror = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
# Introduce t and dt instead of n and p
rror = error.subs('n', 't/dt').subs(p, 'a*dt')
rror = error.as_leading_term(dt) # study only the first term
rint error
rror_L2 = sqrt(integrate(error**2, (t, 0, T)))
rint error_L2
```

he output reads

```
sqrt(30)*sqrt(T**3*a**4*dt**2*(6*T**2*a**2 - 15*T*a + 10))/60
```

hich means that the L^2 error behaves like $a^2\Delta t$.

Strictly speaking, the numerical error is only defined at the mesh points so it akes most sense to compute the ℓ^2 error

$$||e^n||_{\ell^2} = \sqrt{\Delta t \sum_{n=0}^{N_t} (u_{\mathrm{e}}(t_n) - u^n)^2}\,.$$

Ve have obtained an exact analytical expressions for the error at $t = t_n$, but ere we use the leading-order error term only since we are mostly interested in ow the error behaves as a polynomial in Δt, and then the leading order term ill dominate. For the Forward Euler scheme, $u_{\mathrm{e}}(t_n) - u^n \approx \frac{1}{2}np^2$, and we have

$$||e^n||_{\ell^2}^2 = \Delta t \sum_{n=0}^{N_t} \frac{1}{4} n^2 p^4 = \Delta t \frac{1}{4} p^4 \sum_{n=0}^{N_t} n^2\,.$$

ow, $\sum_{n=0}^{N_t} n^2 \approx \frac{1}{3}N_t^3$. Using this approximation, setting $N_t = T/\Delta t$, and aking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{2}\sqrt{\frac{T^3}{3}}a^2\Delta t\,.$$

alculations for the Backward Euler scheme are very similar and provide the ame result, while the Crank-Nicolson scheme leads to

$$||e^n||_{\ell^2} = \frac{1}{12}\sqrt{\frac{T^3}{3}}a^3\Delta t^2\,.$$

Summary of errors.

Both the point-wise and the time-integrated true errors are of second in Δt for the Crank-Nicolson scheme and of first order in Δt for the For Euler and Backward Euler schemes.

3.9 Truncation error

The truncation error is a very frequently used error measure for finite di methods. It is defined as *the error in the difference equation that aris inserting the exact solution*. Contrary to many other error measures, true error $e^n = u_{\mathrm{e}}(t_n) - u^n$, the truncation error is a quantity that i computable.

Let us illustrate the calculation of the truncation error for the Forwa scheme. We start with the difference equation on operator form,

$$[D_t u = -au]^n,$$

i.e.,

$$\frac{u^{n+1} - u^n}{\Delta t} = -au^n\,.$$

The idea is to see how well the exact solution $u_{\mathrm{e}}(t)$ fulfills this equatio $u_{\mathrm{e}}(t)$ in general will not obey the discrete equation, error in the discrete e called a *residual*, denoted here by R^n:

$$R^n = \frac{u_{\mathrm{e}}(t_{n+1}) - u_{\mathrm{e}}(t_n)}{\Delta t} + au_{\mathrm{e}}(t_n)\,.$$

The residual is defined at each mesh point and is therefore a mesh functi a superscript n.

The interesting feature of R^n is to see how it depends on the discre parameter Δt. The tool for reaching this goal is to Taylor expand u_{e} aro point where the difference equation is supposed to hold, here $t = t_n$. V that

$$u_{\mathrm{e}}(t_{n+1}) = u_{\mathrm{e}}(t_n) + u'_{\mathrm{e}}(t_n)\Delta t + \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t^2 + \cdots$$

Inserting this Taylor series in (55) gives

$$R^n = u'_{\mathrm{e}}(t_n) + \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t + \ldots + au_{\mathrm{e}}(t_n)\,.$$

Now, u_{e} fulfills the ODE $u'_{\mathrm{e}} = -au_{\mathrm{e}}$ such that the first and last term and we have

$$R^n \approx \frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t\,.$$

his R^n is the *truncation error*, which for the Forward Euler is seen to be of rst order in Δt.

The above procedure can be repeated for the Backward Euler and the Crank-icolson schemes. We start with the scheme in operator notation, write it out in etail, Taylor expand u_{e} around the point $\tilde{t}$ at which the difference equation is efined, collect terms that correspond to the ODE (here $u'_{\mathrm{e}} + au_{\mathrm{e}}$), and identify ıe remaining terms as the residual R, which is the truncation error. The ackward Euler scheme leads to

$$R^n \approx -\frac{1}{2}u''_{\mathrm{e}}(t_n)\Delta t,$$

hile the Crank-Nicolson scheme gives

$$R^{n+\frac{1}{2}} \approx \frac{1}{24}u'''_{\mathrm{e}}(t_{n+\frac{1}{2}})\Delta t^2\,.$$

The *order* r of a finite difference scheme is often defined through the leading erm Δt^r in the truncation error. The above expressions point out that the orward and Backward Euler schemes are of first order, while Crank-Nicolson of second order. We have looked at other error measures in other sections, ke the error in amplification factor and the error $e^n = u_{\mathrm{e}}(t_n) - u^n$, and xpressed these error measures in terms of Δt to see the order of the method. ormally, calculating the truncation error is more straightforward than deriving ıe expressions for other error measures and therefore the easiest way to establish ıe order of a scheme.

.10 Consistency, stability, and convergence

hree fundamental concepts when solving differential equations by numerical ıethods are consistency, stability, and convergence. We shall briefly touch these oncepts below in the context of the present model problem.

Consistency means that the error in the difference equation, measured through ıe truncation error, goes to zero as $\Delta t \to 0$. Since the truncation error ells how well the exact solution fulfills the difference equation, and the exact olution fulfills the differential equation, consistency ensures that the difference quation approaches the differential equation in the limit. The expressions for the runcation errors in the previous section are all proportional to Δt or Δt^2, hence ıey vanish as $\Delta t \to 0$, and all the schemes are consistent. Lack of consistency nplies that we actually solve a different differential equation in the limit $\Delta t \to 0$ ıan we aim at.

Stability means that the numerical solution exhibits the same qualitative roperties as the exact solution. This is obviously a feature we want the numerical olution to have. In the present exponential decay model, the exact solution is ıonotone and decaying. An increasing numerical solution is not in accordance ith the decaying nature of the exact solution and hence unstable. We can also ay that an oscillating numerical solution lacks the property of monotonicity f the exact solution and is also unstable. We have seen that the Backward

Euler scheme always leads to monotone and decaying solutions, regardles and is hence stable. The Forward Euler scheme can lead to increasing s and oscillating solutions if Δt is too large and is therefore unstable unle sufficiently small. The Crank-Nicolson can never lead to increasing soluti has no problem to fulfill that stability property, but it can produce osc solutions and is unstable in that sense, unless Δt is sufficiently small.

Convergence implies that the global (true) error mesh function $e^n =$ $u^n \to 0$ as $\Delta t \to 0$. This is really what we want: the numerical solution close to the exact solution as we request by having a sufficiently fine m

Convergence is hard to establish theoretically, except in quite simple p like the present one. Stability and consistency are much easier to calcu major breakthrough in the understanding of numerical methods for diff equations came in 1956 when Lax and Richtmeyer established equi between convergence on one hand and consistency and stability on the ot Lax equivalence theorem[18]). In practice it meant that one can first establ a method is stable and consistent, and then it is automatically convergent is much harder to establish). The result holds for linear problems only the world of nonlinear differential equations the relations between cons stability, and convergence are much more complicated.

We have seen in the previous analysis that the Forward Euler, Ba Euler, and Crank-Nicolson schemes are convergent ($e^n \to 0$), that t consistent ($R^n \to 0$, and that they are stable under certain conditions size of Δt. We have also derived explicit mathematical expressions for truncation error, and the stability criteria.

4 Exercises

Exercise 1: Visualize the accuracy of finite differenc e^{-at}

The purpose of this exercise is to visualize the accuracy of finite di approximations of the derivative of a given function. For any finite di approximation, take the Forward Euler difference as an example, and any function, take $u = e^{-at}$, we may introduce an error fraction specific

$$E = \frac{[D_t^+ u]^n}{u'(t_n)} = \frac{\exp\left(-a(t_n + \Delta t)\right) - \exp\left(-at_n\right)}{-a\exp\left(-at_n\right)} = -\frac{1}{a\Delta t}\left(\exp\left(-a\Delta t\right)\right.$$

and view E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while deviate significantly from unit for large Δt. How the error depends on Δ visualized in a graph where we use a logarithmic scale on for Δt, so we c many orders of magnitude of that quantity. Here is a code segment crea array of 100 intervals, on the logarithmic scale, ranging from 10^{-6} to 1 a plotting E versus $p = a\Delta t$ with logarithmic scale on the Δt axis:

[18] `http://en.wikipedia.org/wiki/Lax_equivalence_theorem`

```
from numpy import logspace, exp
from matplotlib.pyplot import plot
p = logspace(-6, 1, 101)
y = -(exp(-p)-1)/p
semilog(p, y)
```

lustrate such errors for the finite difference operators $[D_t^+ u]^n$ (forward), $[D_t^- u]^n$ backward), and $[D_t u]^n$ (centered).

Perform a Taylor series expansions of the error fractions and find the leading rder r in the expressions of type $1+C\Delta t^r+\mathcal{O}(\Delta t^{r+1})$, where C is some constant. ilename: `decay_plot_fd_exp_error.py`.

Exercise 2: Explore the θ-rule for exponential growth

his exercise asks you to solve the ODE $u' = -au$ with $a < 0$ such that the DE models exponential growth instead of exponential decay. A central theme to investigate numerical artifacts and non-physical solution behavior.

) Run experiments with $\theta = 0, 0.5, 1$ for various values of Δt to uncover umerical artifacts. Recall that the exact solution is a monotone, growing nction when $a < 0$. Oscillations or significantly wrong growth are signs of rong qualitative behavior, which can be taken as a stability criterion.

Use the insight to select a few values of Δt and a fixed that demonstrate all ypes of numerical artifacts for the three different schemes ($\theta = 0, 0.5, 1$).

lint. Modify the `decay_exper1.py` code to suit your needs.
ilename: `growth_exper.py`.

) Write up the amplification factor and plot it for $\theta = 0, 0.5, 1$ together with ıe exact one for $a\Delta t < 0$. Use the plot to explain the observations made in the xperiments.

lint. Modify the `decay_ampf_plot.py`[19] code.
ilename: `growth_ampf.py`.

) Write a scientific report about the findings.

lint. Use examples from Section **??** to see how scientific reports can be written.
ilenames: `growth_exper.pdf`, `growth_exper.html`.

Model extensions

is time to consider generalizations of the simple decay model $u = -au$ and lso to look at additional numerical solution methods.

[19] `http://tinyurl.com/jvzzcfn/decay/decay_ampf_plot.py`

5.1 Generalization: including a variable coefficient

In the ODE for decay, $u' = -au$, we now consider the case where a dep time:

$$u'(t) = -a(t)u(t), \quad t \in (0,T], \quad u(0) = I\,.$$

A Forward Euler scheme consist of evaluating (56) at $t = t_n$ and app ing the derivative with a forward difference $[D_t^+ u]^n$:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_n)u^n\,.$$

The Backward Euler scheme becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)u^n\,.$$

The Crank-Nicolson method builds on sampling the ODE at $t_{n+\frac{1}{2}}$. evaluate a at $t_{n+\frac{1}{2}}$ and use an average for u at times t_n and t_{n+1}:

$$\frac{u^{n+1} - u^n}{\Delta t} = -a(t_{n+\frac{1}{2}})\frac{1}{2}(u^n + u^{n+1})\,.$$

Alternatively, we can use an average for the product au:

$$\frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{2}(a(t_n)u^n + a(t_{n+1})u^{n+1})\,.$$

The θ-rule unifies the three mentioned schemes. One version is to evaluated at $t_{n+\theta}$,

$$\frac{u^{n+1} - u^n}{\Delta t} = -a((1-\theta)t_n + \theta t_{n+1})((1-\theta)u^n + \theta u^{n+1})\,.$$

Another possibility is to apply a weighted average for the product au,

$$\frac{u^{n+1} - u^n}{\Delta t} = -(1-\theta)a(t_n)u^n - \theta a(t_{n+1})u^{n+1}\,.$$

With the finite difference operator notation the Forward Euler and B Euler schemes can be summarized as

$$[D_t^+ u = -au]^n,$$
$$[D_t^- u = -au]^n\,.$$

The Crank-Nicolson and θ schemes depend on whether we evaluate sample point for the ODE or if we use an average. The various versi written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}, \tag{65}$$
$$[D_t u = -\overline{au}^t]^{n+\frac{1}{2}}, \tag{66}$$
$$[D_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \tag{67}$$
$$[D_t u = -\overline{au}^{t,\theta}]^{n+\theta}\,. \tag{68}$$

.2 Generalization: including a source term

further extension of the model ODE is to include a source term $b(t)$:

$$u'(t) = -a(t)u(t) + b(t), \quad t \in (0,T], \quad u(0) = I\,. \tag{69}$$

chemes. The time point where we sample the ODE determines where $b(t)$ is valuated. For the Crank-Nicolson scheme and the θ-rule we have a choice of hether to evaluate $a(t)$ and $b(t)$ at the correct point or use an average. The osen strategy becomes particularly clear if we write up the schemes in the perator notation:

$$[D_t^+ u = -au + b]^n, \tag{70}$$
$$[D_t^- u = -au + b]^n, \tag{71}$$
$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{72}$$
$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}}, \tag{73}$$
$$[D_t u = -a\overline{u}^{t,\theta} + b]^{n+\theta}, \tag{74}$$
$$[D_t u = \overline{-au + b}^{t,\theta}]^{n+\theta}\,. \tag{75}$$

.3 Implementation of the generalized model problem

)eriving the θ-rule formula. Writing out the θ-rule in (75), using (32) and 3), we get

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta(-a^{n+1}u^{n+1} + b^{n+1})) + (1-\theta)(-a^n u^n + b^n)), \tag{76}$$

here a^n means evaluating a at $t = t_n$ and similar for a^{n+1}, b^n, and b^{n+1}. We olve for u^{n+1}:

$$u^{n+1} = ((1 - \Delta t(1-\theta)a^n)u^n + \Delta t(\theta b^{n+1} + (1-\theta)b^n))(1 + \Delta t\theta a^{n+1})^{-1}\,. \tag{77}$$

'he Python code. Here is a suitable implementation of (76) where $a(t)$ and (t) are given as Python functions:


```
def solver(I, a, b, T, dt, theta):
    """
    Solve u'=-a(t)*u + b(t), u(0)=I,
    for t in (0,T] with steps of dt.
    a and b are Python functions of t.
    """
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = ((1 - dt*(1-theta)*a(t[n]))*u[n] + \
                  dt*(theta*b(t[n+1]) + (1-theta)*b(t[n])))/\
                  (1 + dt*theta*a(t[n+1]))
    return u, t
```

This function is found in the file `decay_vc.py`[20] (`vc` stands for "variab cients").

Coding of variable coefficients. The `solver` function shown above c the arguments `a` and `b` to be Python functions of time `t`, say

```
def a(t):
    return a_0 if t < tp else k*a_0

def b(t):
    return 1
```

Here, `a(t)` has three parameters `a0`, `tp`, and `k`, which must be global v A better implementation is to represent `a` by a class where the parame attributes and a *special method* `__call__` evaluates $a(t)$:

```
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k

    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0

a = A(a0=2, k=1)  # a behaves as a function a(t)
```

For quick tests it is cumbersome to write a complete function or a cla *lambda function* construction in Python is then convenient. For exampl

```
a = lambda t: a_0 if t < tp else k*a_0
```

is equivalent to the `def a(t):` definition above. In general,

[20] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

```
f = lambda arg1, arg2, ...: expressin
```

equivalent to

```
def f(arg1, arg2, ...):
    return expression
```

ne can use lambda functions directly in calls. Say we want to solve $u' = -u+1$, $(0) = 2$:

```
u, t = solver(2, lambda t: 1, lambda t: 1, T, dt, theta)
```

lambda function can appear anywhere where a variable can appear.

.4 Verifying a constant solution

very useful partial verification method is to construct a test problem with very simple solution, usually $u = \text{const}$. Especially the initial debugging of program code can benefit greatly from such tests, because 1) all relevant umerical methods will exactly reproduce a constant solution, 2) many of the ntermediate calculations are easy to control for a constant u, and 3) even a onstant u can uncover many bugs in an implementation.

The only constant solution for the problem $u' = -au$ is $u = 0$, but too many ugs can escape from that trivial solution. It is much better to search for a roblem where $u = C = \text{const} \neq 0$. Then $u' = -a(t)u + b(t)$ is more appropriate: ith $u = C$ we can choose any $a(t)$ and set $b = a(t)C$ and $I = C$. An appropriate ose test is

```
import nose.tools as nt

def test_constant_solution():
    """
    Test problem where u=u_const is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return u_const

    def a(t):
        return 2.5*(1+t**3)  # can be arbitrary

    def b(t):
        return a(t)*u_const

    u_const = 2.15
    theta = 0.4; I = u_const; dt = 4
    Nt = 4  # enough with a few steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    print u
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    nt.assert_almost_equal(difference, 0, places=14)
```


An interesting question is what type of bugs that will make the comp deviate from the exact solution C. Fortunately, the updating formula initial condition must be absolutely correct for the test to pass! Any att make a wrong indexing in terms like `a(t[n])` or any attempt to intro erroneous factor in the formula creates a solution that is different from

5.5 Verification via manufactured solutions

Following the idea of the previous section, we can choose any formul exact solution, insert the formula in the ODE problem and fit the data a and I to make the chosen formula fulfill the equation. This powerful te for generating exact solutions is very useful for verification purposes anc as the *method of manufactured solutions*, often abbreviated MMS.

One common choice of solution is a linear function in the inde variable(s). The rationale behind such a simple variation is that alm relevant numerical solution method for differential equation problems is reproduce the linear function exactly to machine precision (if u is abo in size; precision is lost if u take on large values, see Exercise 3). Th solution also makes some stronger demands to the numerical method implementation than the constant solution used in Section 5.4, at least complicated applications. However, the constant solution is often ideal fc debugging before proceeding with a linear solution.

We choose a linear solution $u(t) = ct + d$. From the initial condition it that $d = I$. Inserting this u in the ODE results in

$$c = -a(t)u + b(t)\,.$$

Any function $u = ct + I$ is then a correct solution if we choose

$$b(t) = c + a(t)(ct + I)\,.$$

With this $b(t)$ there are no restrictions on $a(t)$ and c.

Let prove that such a linear solution obeys the numerical schemes. end, we must check that $u^n = ca(t_n)(ct_n + I)$ fulfills the discrete equatic these calculations, and later calculations involving linear solutions ins finite difference schemes, it is convenient to compute the action of a di operator on a linear function t:

$$[D_t^+ t]^n = \frac{t_{n+1} - t_n}{\Delta t} = 1,$$
$$[D_t^- t]^n = \frac{t_n - t_{n-1}}{\Delta t} = 1,$$
$$[D_t t]^n = \frac{t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}}}{\Delta t} = \frac{(n+\frac{1}{2})\Delta t - (n-\frac{1}{2})\Delta t}{\Delta t} = 1\,.$$

Clearly, all three finite difference approximations to the derivative are e $u(t) = t$ or its mesh function counterpart $u^n = t_n$.

The difference equation for the Forward Euler scheme

$$[D_t^+ u = -au + b]^n,$$

ith $a^n = a(t_n)$, $b^n = c + a(t_n)(ct_n + I)$, and $u^n = ct_n + I$ then results in

$$c = -a(t_n)(ct_n + I) + c + a(t_n)(ct_n + I) = c$$

hich is always fulfilled. Similar calculations can be done for the Backward uler and Crank-Nicolson schemes, or the θ-rule for that matter. In all cases, $^n = ct_n + I$ is an exact solution of the discrete equations. That is why we ıould expect that $u^n - u_{\mathrm{e}}(t_n) = 0$ mathematically and $|u^n - u_{\mathrm{e}}(t_n)|$ less than small number about the machine precision for $n = 0, \ldots, N_t$.

The following function offers an implementation of this verification test based n a linear exact solution:

```
def test_linear_solution():
    """
    Test problem where u=c*t+I is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return c*t + I

    def a(t):
        return t**0.5  # can be arbitrary

    def b(t):
        return c + a(t)*exact_solution(t)

    theta = 0.4; I = 0.1; dt = 0.1; c = -0.5
    T = 4
    Nt = int(T/dt)  # no of steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    print difference
    # No of decimal places for comparison depend on size of c
    nt.assert_almost_equal(difference, 0, places=14)
```

ny error in the updating formula makes this test fail!

Choosing more complicated formulas as the exact solution, say $\cos(t)$, will not ıake the numerical and exact solution coincide to machine precision, because nite differencing of $\cos(t)$ does not exactly yield the exact derivative $-\sin(t)$. In ıch cases, the verification procedure must be based on measuring the convergence ıtes as exemplified in Section **??**. Convergence rates can be computed as long s one has an exact solution of a problem that the solver can be tested on, but ıis can always be obtained by the method of manufactured solutions.

.6 Extension to systems of ODEs

Iany ODE models involves more than one unknown function and more than ne equation. Here is an example of two unknown functions $u(t)$ and $v(t)$:

$$u' = au + bv,$$
$$v' = cu + dv,$$

for constants a, b, c, d. Applying the Forward Euler method to each e results in simple updating formula

$$u^{n+1} = u^n + \Delta t(au^n + bv^n),$$
$$v^{n+1} = u^n + \Delta t(cu^n + dv^n)\,.$$

On the other hand, the Crank-Nicolson or Backward Euler schemes res 2×2 linear system for the new unknowns. The latter schemes gives

$$u^{n+1} = u^n + \Delta t(au^{n+1} + bv^{n+1}),$$
$$v^{n+1} = v^n + \Delta t(cu^{n+1} + dv^{n+1})\,.$$

Collecting u^{n+1} as well as v^{n+1} on the left-hand side results in

$$(1 - \Delta ta)u^{n+1} + bv^{n+1} = u^n,$$
$$cu^{n+1} + (1 - \Delta td)v^{n+1} = v^n,$$

which is a system of two coupled, linear, algebraic equations in two unl

6 General first-order ODEs

We now turn the attention to general, nonlinear ODEs and systems ODEs. Our focus is on numerical methods that can be readily reused f discretization PDEs, and diffusion PDEs in particular. The methods briefly listed, and we refer to the rich literature for more detailed desc and analysis - the books [6, 1, 2, 3] are all excellent resources on nu methods for ODEs. We also demonstrate the Odespy Python interfa range of different software for general first-order ODE systems.

6.1 Generic form

ODEs are commonly written in the generic form

$$u' = f(u, t), \quad u(0) = I,$$

where $f(u, t)$ is some prescribed function. As an example, our most exponential decay model (69) has $f(u, t) = -a(t)u(t) + b(t)$.

The unknown u in (89) may either be a scalar function of time t, or valued function of t in case of a *system of ODEs* with m unknown com

$$u(t) = (u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t))\,.$$

ı that case, the right-hand side is vector-valued function with m components,

$$f(u,t) = (f^{(0)}(u^{(0)}(t),\ldots,u^{(m-1)}(t)),\\ f^{(1)}(u^{(0)}(t),\ldots,u^{(m-1)}(t)),\\ \vdots,\\ f^{(m-1)}(u^{(0)}(t),\ldots,u^{(m-1)}(t)))\,.$$

Actually, any system of ODEs can be written in the form (89), but higher-rder ODEs then need auxiliary unknown functions to enable conversion to a rst-order system.

Next we list some well-known methods for $u' = f(u,t)$, valid both for a ngle ODE (scalar u) and systems of ODEs (vector u). The choice of methods inspired by the kind of schemes that are popular also for partial differential quations.

.2 The θ-rule

he θ-rule scheme applied to $u' = f(u,t)$ becomes

$$\frac{u^{n+1}-u^n}{\Delta t} = \theta f(u^{n+1},t_{n+1}) + (1-\theta)f(u^n,t_n)\,. \tag{90}$$

ringing the unknown u^{n+1} to the left-hand side and the known terms on the ght-hand side gives

$$u^{n+1} - \Delta t\theta f(u^{n+1},t_{n+1}) = u^n + \Delta t(1-\theta)f(u^n,t_n)\,. \tag{91}$$

or a general f (not linear in u), this equation is *nonlinear* in the unknown u^{n+1} nless $\theta = 0$. For a scalar ODE ($m = 1$), we have to solve a single nonlinear lgebraic equation for u^{n+1}, while for a system of ODEs, we get a system of oupled, nonlinear algebraic equations. Newton's method is a popular solution pproach in both cases. Note that with the Forward Euler scheme ($\theta = 0$) we o not have to deal with nonlinear equations, because in that case we have an xplicit updating formula for u^{n+1}. This is known as an *explicit* scheme. With $\neq 1$ we have to solve systems of algebraic equations, and the scheme is said to e *implicit*.

.3 An implicit 2-step backward scheme

he implicit backward method with 2 steps applies a three-level backward ifference as approximation to $u'(t)$,

$$u'(t_{n+1}) \approx \frac{3u^{n+1}-4u^n+u^{n-1}}{2\Delta t},$$

which is an approximation of order Δt^2 to the first derivative. The r scheme for $u' = f(u,t)$ reads

$$u^{n+1} = \frac{4}{3}u^n - \frac{1}{3}u^{n-1} + \frac{2}{3}\Delta t f(u^{n+1},t_{n+1})\,.$$

Higher-order versions of the scheme (92) can be constructed by includi time levels. These schemes are known as the Backward Differentiation F (BDF), and the particular version (92) is often referred to as BDF2.

Note that the scheme (92) is implicit and requires solution of n equations when f is nonlinear in u. The standard 1st-order Backwar method or the Crank-Nicolson scheme can be used for the first step.

6.4 Leapfrog schemes

The ordinary Leapfrog scheme. The derivative of u at some poin be approximated by a central difference over two time steps,

$$u'(t_n) \approx \frac{u^{n+1}-u^{n-1}}{2\Delta t} = [D_{2t}u]^n$$

which is an approximation of second order in Δt. The scheme can written as

$$[D_{2t}u = f(u,t)]^n,$$

in operator notation. Solving for u^{n+1} gives

$$u^{n+1} = u^{n-1} + \Delta t f(u^n,t_n)\,.$$

Observe that (94) is an explicit scheme, and that a nonlinear f (in u) i to handle since it only involves the known u^n value. Some other schen be used as starter to compute u^1, preferably the Forward Euler scheme is also explicit.

The filtered Leapfrog scheme. Unfortunately, the Leapfrog sche will develop growing oscillations with time (see Problem 8)[[[. A rem such undesired oscillations is to introduce a *filtering technique*. First, a s Leapfrog step is taken, according to (94), and then the previous u^n adjusted according to

$$u^n \;\leftarrow\; u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})\,.$$

The γ-terms will effectively damp oscillations in the solution, especiall with short wavelength (like point-to-point oscillations). A common choic 0.6 (a value used in the famous NCAR Climate Model).

.5 The 2nd-order Runge-Kutta scheme

he two-step scheme

$$u^* = u^n + \Delta t f(u^n, t_n), \tag{96}$$

$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(f(u^n, t_n) + f(u^*, t_{n+1})\right), \tag{97}$$

ssentially applies a Crank-Nicolson method (97) to the ODE, but replaces the erm $f(u^{n+1}, t_{n+1})$ by a prediction $f(u^*, t_{n+1})$ based on a Forward Euler step 96). The scheme (96)-(97) is known as Huen's method, but is also a 2nd-order unge-Kutta method. The scheme is explicit, and the error is expected to behave s Δt^2.

.6 A 2nd-order Taylor-series method

ne way to compute u^{n+1} given u^n is to use a Taylor polynomial. We may write p a polynomial of 2nd degree:

$$u^{n+1} = u^n + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2\,.$$

rom the equation $u' = f(u,t)$ it follows that the derivatives of u can be expressed ı terms of f and its derivatives:

$$\begin{aligned} u'(t_n) &= f(u^n, t_n), \\ u''(t_n) &= \frac{\partial f}{\partial u}(u^n, t_n)u'(t_n) + \frac{\partial f}{\partial t} \\ &= f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}, \end{aligned}$$

esulting in the scheme

$$u^{n+1} = u^n + f(u^n, t_n)\Delta t + \frac{1}{2}\left(f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}\right)\Delta t^2\,. \tag{98}$$

Iore terms in the series could be included in the Taylor polynomial to obtain ıethods of higher order than 2.

.7 The 2nd- and 3rd-order Adams-Bashforth schemes

he following method is known as the 2nd-order Adams-Bashforth scheme:

$$u^{n+1} = u^n + \frac{1}{2}\Delta t\left(3f(u^n, t_n) - f(u^{n-1}, t_{n-1})\right)\,. \tag{99}$$

he scheme is explicit and requires another one-step scheme to compute u^1 (the orward Euler scheme or Heun's method, for instance). As the name implies, ıe scheme is of order Δt^2.

Another explicit scheme, involving four time levels, is the 3rd-order Bashforth scheme

$$u^{n+1} = u^n + \frac{1}{12}\left(23f(u^n, t_n) - 16f(u^{n-1}, t_{n-1}) + 5f(u^{n-2}, t_{n-2})\right)\,.$$

The numerical error is of order Δt^3, and the scheme needs some met computing u^1 and u^2.

More general, higher-order Adams-Bashforth schemes (also called *Adams methods*) compute u^{n+1} as a linear combination of f at k previc steps:

$$u^{n+1} = u^n + \sum_{j=0}^{k}\beta_j f(u^{n-j}, t_{n-j}),$$

where β_j are known coefficients.

6.8 4th-order Runge-Kutta scheme

The perhaps most widely used method to solve ODEs is the 4th-order Kutta method, often called RK4. Its derivation is a nice illustration of c numerical approximation strategies, so let us go through the steps in d

The starting point is to integrate the ODE $u' = f(u,t)$ from t_n to t

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u(t), t)dt\,.$$

We want to compute $u(t_{n+1})$ and regard $u(t_n)$ as known. The task is good approximations for the integral, since the integrand involves the u u between t_n and t_{n+1}.

The integral can be approximated by the famous Simpson's rule[21]:

$$\int_{t_n}^{t_{n+1}} f(u(t), t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+\frac{1}{2}} + f^{n+1}\right)\,.$$

The problem now is that we do not know $f^{n+\frac{1}{2}} = f(u^{n+\frac{1}{2}}, t_{n+1/2})$ and (u^{n+1}, t_{n+1}) as we know only u^n and hence f^n. The idea is to use various imations for $f^{n+\frac{1}{2}}$ and f^{n+1} based on using well-known schemes for the the intervals $[t_n, t_{n+1/2}]$ and $[t_n, t_{n+1}]$. We split the integral approximat four terms:

$$\int_{t_n}^{t_{n+1}} f(u(t), t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

[21] http://en.wikipedia.org/wiki/Simpson's_rule

here $\hat{f}^{n+\frac{1}{2}}$, $\tilde{f}^{n+\frac{1}{2}}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+\frac{1}{2}}$ and f^{n+1} that can be ased on already computed quantities. For $\hat{f}^{n+\frac{1}{2}}$ we can apply an approximation o $u^{n+\frac{1}{2}}$ using the Forward Euler method with step $\frac{1}{2}\Delta t$:

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+1/2}) \tag{101}$$

ince this gives us a prediction of $f^{n+\frac{1}{2}}$, we can for $\tilde{f}^{n+\frac{1}{2}}$ try a Backward Euler ıethod to approximate $u^{n+\frac{1}{2}}$:

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1/2})\,. \tag{102}$$

/ith $\tilde{f}^{n+\frac{1}{2}}$ as a hopefully good approximation to $f^{n+\frac{1}{2}}$, we can for the final erm $\bar{f}^{n+1}$ use a Crank-Nicolson method to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1})\,. \tag{103}$$

/e have now used the Forward and Backward Euler methods as well as the rank-Nicolson method in the context of Simpson's rule. The hope is that the ombination of these methods yields an overall time-stepping scheme from t_n to $_+1$ that is much more accurate than the $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta t^2)$ of the individual eps. This is indeed true: the overall accuracy is $\mathcal{O}(\Delta t^4)$!

To summarize, the 4th-order Runge-Kutta method becomes

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right), \tag{104}$$

here the quantities on the right-hand side are computed from (101)-(103). ote that the scheme is fully explicit so there is never any need to solve linear r nonlinear algebraic equations. However, the stability is conditional and epends on f. There is a whole range of *implicit* Runge-Kutta methods that are nconditionally stable, but require solution of algebraic equations involving f at ach time step.

The simplest way to explore more sophisticated methods for ODEs is to pply one of the many high-quality software packages that exist, as the next ection explains.

.9 The Odespy software

wide range of the methods and software exist for solving (89). Many of methods re accessible through a unified Python interface offered by the Odespy[22] package. desp features simple Python implementations of the most fundamental schemes s well as Python interfaces to several famous packages for solving ODEs:

[22] https://github.com/hplgit/odespy

ODEPACK[23], Vode[24], rkc.f[25], rkf45.f[26], Radau5[27], as well as the ODE in SciPy[28], SymPy[29], and odelab[30].

The usage of Odespy follows this setup for the ODE $u' = -au$, u $t \in (0, T]$, here solved by the famous 4th-order Runge-Kutta metho $\Delta t = 1$ and $N_t = 6$ steps:

```
def f(u, t):
    return -a*u

import odespy
import numpy as np

I = 1; a = 0.5; Nt = 6; dt = 1
solver = odespy.RK4(f)
solver.set_initial_condition(I)
t_mesh = np.linspace(0, Nt*dt, Nt+1)
u, t = solver.solve(t_mesh)
```

The previously listed methods for ODEs are all accessible in Odesp

- the θ-rule: `ThetaRule`
- special cases of the θ-rule: `ForwardEuler`, `BackwardEuler`, `CrankN`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`
- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode` `lsoda_scipy`
- The Leapfrog scheme: `Leapfrog` and `LeapfrogFiltered`

6.10 Example: Runge-Kutta methods

Since all solvers have the same interface in Odespy, modulo differen parameters to the solvers' constructors, one can easily make a list o objects and run a loop for comparing (a lot of) solvers. The code below in complete form in `decay_odespy.py`[31], compares the famous Rung methods of orders 2, 3, and 4 with the exact solution of the decay e $u' = -au$. Since we have quite long time steps, we have included t relevant θ-rule for large time steps, the Backward Euler scheme ($\theta = 1$), Figure 15 shows the results.

[23] https://computation.llnl.gov/casc/odepack/odepack_home.html
[24] https://computation.llnl.gov/casc/odepack/odepack_home.html
[25] http://www.netlib.org/ode/rkc.f
[26] http://www.netlib.org/ode/rkf45.f
[27] http://www.unige.ch/ hairer/software.html
[28] http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.od
[29] http://docs.sympy.org/dev/modules/mpmath/calculus/odes.html
[30] http://olivierverdier.github.com/odelab/
[31] http://tinyurl.com/jvzzcfn/decay/decay_odespy.py

```
import numpy as np
import scitools.std as plt
import sys

def f(u, t):
    return -a*u

I = 1; a = 2; T = 6
dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.75
Nt = int(round(T/dt))
t = np.linspace(0, Nt*dt, Nt+1)

solvers = [odespy.RK2(f),
           odespy.RK3(f),
           odespy.RK4(f),
           odespy.BackwardEuler(f, nonlinear_solver='Newton')]

legends = []
for solver in solvers:
    solver.set_initial_condition(I)
    u, t = solver.solve(t)

    plt.plot(t, u)
    plt.hold('on')
    legends.append(solver.__class__.__name__)

# Compare with exact solution plotted on a very fine mesh
t_fine = np.linspace(0, T, 10001)
u_e = I*np.exp(-a*t_fine)
plt.plot(t_fine, u_e, '-') # avoid markers by specifying line type
legends.append('exact')

plt.legend(legends)
plt.title('Time step: %g' % dt)
plt.show()
```

Visualization tip.

We use SciTools for plotting here, but importing `matplotlib.pyplot` as `plt` instead also works. However, plain use of Matplotlib as done here results in curves with different colors, which may be hard to distinguish on black-and-white paper. Using SciTools, curves are automatically given colors *and* markers, thus making curves easy to distinguish on screen with colors and on black-and-white paper. The automatic adding of markers is normally a bad idea for a very fine mesh since all the markers get cluttered, but SciTools limits the number of markers in such cases. For the exact solution we use a very fine mesh, but in the code above we specify the line type as a solid line (`-`), which means no markers and just a color to be automatically determined by the backend used for plotting (Matplotlib by default, but SciTools gives the opportunity to use other backends to produce the plot, e.g., Gnuplot or Grace).

Also note the that the legends are based on the class names o solvers, and in Python the name of a the class type (as a string) object `obj` is obtained by `obj.__class__.__name__`.

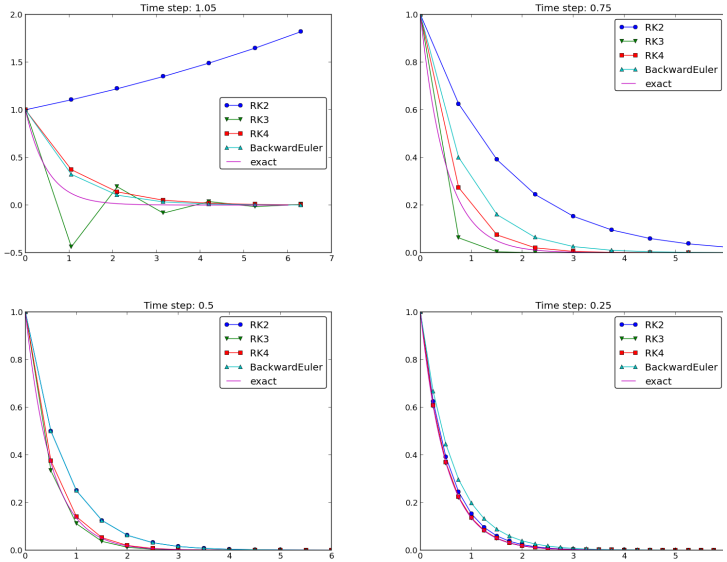


Figure 15: Behavior of different schemes for the decay equation.

The runs in Figure 15 and other experiments reveal that the 2n Runge-Kutta method (`RK2`) is unstable for $\Delta t > 1$ and decays slower t Backward Euler scheme for large and moderate Δt (see Exercise 7 for an a However, for fine $\Delta t = 0.25$ the 2nd-order Runge-Kutta method app the exact solution faster than the Backward Euler scheme. That is, th scheme does a better job for larger Δt, while the higher order scheme is for smaller Δt. This is a typical trend also for most schemes for ordin partial differential equations.

The 3rd-order Runge-Kutta method (`RK3`) has also artifacts in oscillatory behavior for the larger Δt values, much like that of the Nicolson scheme. For finer Δt, the 3rd-order Runge-Kutta method co quickly to the exact solution.

The 4th-order Runge-Kutta method (`RK4`) is slightly inferior to the Ba Euler scheme on the coarsest mesh, but is then clearly superior to all th schemes. It is definitely the method of choice for all the tested schemes

Remark about using the θ-rule in Odespy. The Odespy package that the ODE is written as $u' = f(u,t)$ with an f that is possibly nonlin

he θ-rule for $u' = f(u,t)$ leads to

$$u^{n+1} = u^n + \Delta t \left(\theta f(u^{n+1}, t_{n+1}) + (1-\theta) f(u^n, t_n)\right),$$

hich is a *nonlinear equation* in u^{n+1}. Odespy's implementation of the θ-rule `hetaRule`) and the specialized Backward Euler (`BackwardEuler`) and Crank-icolson (`CrankNicolson`) schemes must invoke iterative methods for solving ıe nonlinear equation in u^{n+1}. This is done even when f is linear in u, as ı the model problem $u' = -au$, where we can easily solve for u^{n+1} by hand. herefore, we need to specify use of Newton's method to the equations. (Odespy llows other methods than Newton's to be used, for instance Picard iteration, ut that method is not suitable. The reason is that it applies the Forward Euler :heme to generate a start value for the iterations. Forward Euler may give very rong solutions for large Δt values. Newton's method, on the other hand, is ısensitive to the start value in *linear problems*.)

.11 Example: Adaptive Runge-Kutta methods

)despy offers solution methods that can adapt the size of Δt with time to match desired accuracy in the solution. Intuitively, small time steps will be chosen in reas where the solution is changing rapidly, while larger time steps can be used here the solution is slowly varying. Some kind of *error estimator* is used to djust the next time step at each time level.

A very popular adaptive method for solving ODEs is the Dormand-Prince unge-Kutta method of order 4 and 5. The 5th-order method is used as a eference solution and the difference between the 4th- and 5th-order methods is sed as an indicator of the error in the numerical solution. The Dormand-Prince ıethod is the default choice in MATLAB's widely used `ode45` routine.

We can easily set up Odespy to use the Dormand-Prince method and see ow it selects the optimal time steps. To this end, we request only one time step om $t = 0$ to $t = T$ and ask the method to compute the necessary non-uniform me mesh to meet a certain error tolerance. The code goes like

```
import odespy
import numpy as np
import decay_mod
import sys
#import matplotlib.pyplot as plt
import scitools.std as plt

def f(u, t):
    return -a*u

def exact_solution(t):
    return I*np.exp(-a*t)

I = 1; a = 2; T = 5
tol = float(sys.argv[1])
solver = odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

Nt = 1  # just one step - let the scheme find its intermediate points
```



```
t_mesh = np.linspace(0, T, Nt+1)
t_fine = np.linspace(0, T, 10001)

solver.set_initial_condition(I)
u, t = solver.solve(t_mesh)

# u and t will only consist of [I, u^Nt] and [0,T]
# solver.u_all and solver.t_all contains all computed points
plt.plot(solver.t_all, solver.u_all, 'ko')
plt.hold('on')
plt.plot(t_fine, exact_solution(t_fine), 'b-')
plt.legend(['tol=%.0E' % tol, 'exact'])
plt.savefig('tmp_odespy_adaptive.png')
plt.show()
```

Running four cases with tolerances 10^{-1}, 10^{-3}, 10^{-5}, and 10^{-7}, g results in Figure 16. Intuitively, one would expect denser points in the be of the decay and larger time steps when the solution flattens out.

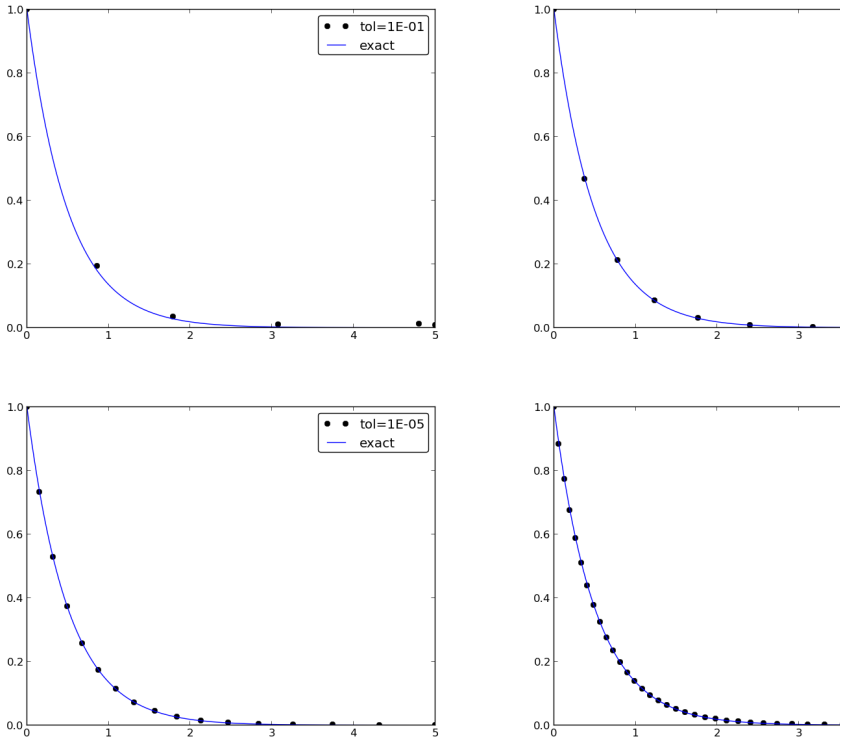


Figure 16: Choice of adaptive time mesh by the Dormand-Prince met different tolerances.

Exercises

Exercise 3: Experiment with precision in tests and the size of u

It is claimed in Section 5.5 that most numerical methods will reproduce a linear exact solution to machine precision. Test this assertion using the nose test function `test_linear_solution` in the `decay_vc.py`[32] program. Vary the parameter `c` from very small, via `c=1` to many larger values, and print out the maximum difference between the numerical solution and the exact solution. What is the relevant value of the `places` (or `delta`) argument to `nose.tools.assert_almost_equal` in each case? Filename: `test_precision.py`.

Exercise 4: Implement the 2-step backward scheme

Implement the 2-step backward method (92) for the model $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. Allow the first step to be computed by either the Backward Euler scheme or the Crank-Nicolson scheme. Verify the implementation by choosing $a(t)$ and $b(t)$ such that the exact solution is linear in t (see Section 5.5). Show mathematically that a linear solution is indeed a solution of the discrete equations.

Compute convergence rates (see Section **??**) in a test case $a = \text{const}$ and $b = 0$, where we easily have an exact solution, and determine if the choice of first-order scheme (Backward Euler) for the first step has any impact on the overall accuracy of this scheme. The expected error goes like $\mathcal{O}(\Delta t^2)$. Filename: `decay_backward2step.py`.

Exercise 5: Implement the 2nd-order Adams-Bashforth scheme

Implement the 2nd-order Adams-Bashforth method (99) for the decay problem $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Use the Forward Euler method for the first step such that the overall scheme is explicit. Verify the implementation using an exact solution that is linear in time. Analyze the scheme by searching for solutions $u^n = A^n$ when $a = \text{const}$ and $b = 0$. Compare this second-order scheme to the Crank-Nicolson scheme. Filename: `decay_AdamsBashforth2.py`.

Exercise 6: Implement the 3rd-order Adams-Bashforth scheme

Implement the 3rd-order Adams-Bashforth method (100) for the decay problem $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Since the scheme is explicit, allow it to be started by two steps with the Forward Euler method. Investigate experimentally the case where $b = 0$ and a is a constant: Can we have oscillatory solutions for large Δt? Filename: `decay_AdamsBashforth3.py`.

[32] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

Exercise 7: Analyze explicit 2nd-order methods

Show that the schemes (97) and (98) are identical in the case $f(u, t) = -a$ where $a > 0$ is a constant. Assume that the numerical solution reads $u^n = A^n$ for unknown amplification factor A to be determined. Find A and derive stability criteria. Can the scheme produce oscillatory solutions of $u' = -au$? Plot the numerical and exact amplification factor. Filename: `decay_RK2_Taylor`.

Problem 8: Implement and investigate the Leapfrog scheme

A Leapfrog scheme for the ODE $u'(t) = -a(t)u(t) + b(t)$ is defined by

$$[D_{2t}u = -au + b]^n .$$

A separate method is needed to compute u^1. The Forward Euler scheme is a possible candidate.

a) Implement the Leapfrog scheme for the model equation. Plot the solution in the case $a = 1$, $b = 0$, $I = 1$, $\Delta t = 0.01$, $t \in [0, 4]$. Compare with the solution $u_{\mathrm{e}}(t) = e^{-t}$.

b) Show mathematically that a linear solution in t fulfills the Forward scheme for the first step and the Leapfrog scheme for the subsequent steps. Use this linear solution to verify the implementation, and automate the verification through a nose test.

Hint. It can be wise to automate the calculations such that it is easy to change the calculations for other types of solutions. Here is a possible `sympy` function that takes a symbolic expression `u` (implemented as a Python function of `t`) and the `b` term, and checks if `u` fulfills the discrete equations:

```
import sympy as sp

def analyze(u):
    t, dt, a = sp.symbols('t dt a')

    print 'Analyzing u_e(t)=%s' % u(t)
    print 'u(0)=%s' % u(t).subs(t, 0)

    # Fit source term to the given u(t)
    b = sp.diff(u(t), t) + a*u(t)
    b = sp.simplify(b)
    print 'Source term b:', b

    # Residual in discrete equations; Forward Euler step
    R_step1 = (u(t+dt) - u(t))/dt + a*u(t) - b
    R_step1 = sp.simplify(R_step1)
    print 'Residual Forward Euler step:', R_step1

    # Residual in discrete equations; Leapfrog steps
    R = (u(t+dt) - u(t-dt))/(2*dt) + a*u(t) - b
    R = sp.simplify(R)
    print 'Residual Leapfrog steps:', R
```

```
def u_e(t):
    return c*t + I

analyze(u_e)
# or short form: analyze(lambda t: c*t + I)
```

) Show that a second-order polynomial in t cannot be a solution of the discrete quations. However, if a Crank-Nicolson scheme is used for the first step, a econd-order polynomial solves the equations exactly.

) Create a manufactured solution $u(t) = \sin(t)$ for the ODE $u' = -au + b$. ompute the convergence rate of the Leapfrog scheme using this manufactured olution. The expected convergence rate of the Leapfrog scheme is $\mathcal{O}(\Delta t^2)$. Does ne use of a 1st-order method for the first step impact the convergence rate?

) Set up a set of experiments to demonstrate that the Leapfrog scheme (94) is ssociated with numerical artifacts (instabilities). Document the main results om this investigation.

) Analyze and explain the instabilities of the Leapfrog scheme (94):

1. Choose $a = \text{const}$ and $b = 0$. Assume that an exact solution of the discrete equations has the form $u^n = A^n$, where A is an amplification factor to be determined. Derive an equation for A by inserting $u^n = A^n$ in the Leapfrog scheme.
2. Compute A either by hand and/or with the aid of `sympy`. The polynomial for A has two roots, A_1 and A_2. Let u^n be a linear combination $u^n = C_1 A_1^n + C_2 A_2^n$.
3. Show that one of the roots is the explanation of the instability.
4. Compare A with the exact expression, using a Taylor series approximation.
5. How can C_1 and C_2 be determined?

) Since the original Leapfrog scheme is unconditionally unstable as time grows, demands some stabilization. This can be done by filtering, where we first find $^{n+1}$ from the original Leapfrog scheme and then replace u^n by $u^n + \gamma(u^{n-1} - u^n + u^{n+1})$, where γ can be taken as 0.6. Implement the filtered Leapfrog cheme and check that it can handle tests where the original Leapfrog scheme is nstable.
ilenames: `decay_leapfrog.py`, `decay_leapfrog.pdf`.

Problem 9: Make a unified implementation of many sc

Consider the linear ODE problem $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. schemes for this problem can be written in the general form

$$u^{n+1} = \sum_{j=0}^{m} c_j u^{n-j},$$

for some choice of $c_0, \ldots, c_m$. Find expressions for the c_j coefficients in the θ-rule, the three-level backward scheme, the Leapfrog scheme, the 2r Runge-Kutta method, and the 3rd-order Adams-Bashforth scheme.

Make a class `ExpDecay` that implements the general updating formu The formula cannot be applied for $n < m$, and for those n values, other must be used. Assume for simplicity that we just repeat Crank-Nicols until (105) can be used. Use a subclass to specify the list $c_0, \ldots, c$ particular method, and implement subclasses for all the mentioned s Verify the implementation by testing with a linear solution, which sh exactly reproduced by all methods. Filename: `decay_schemes_oo.py`.

8 Applications of exponential decay model

This section presents many mathematical models that all end up with (the type $u' = -au + b$. The applications are taken from biology, finar physics, and cover population growth or decay, compound interest and i radioactive decay, cooling of objects, compaction of geological media, p variations in the atmosphere, and air resistance on falling or rising bod

8.1 Scaling

Real applications of a model $u' = -au + b$ will often involve a lot of par in the expressions for a and b. It can be quite a challenge to find relevan of all parameters. In simple problems, however, it turns out that it is not necessary to estimate all parameters because we can lump them into (few *dimensionless* numbers by using a very attractive technique called sc simply means to stretch the u and t axis is the present problem - and sudc parameters in the problem are lumped one parameter if $b \neq 0$ and no pa when $b = 0$!

Scaling means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

where u_m is a characteristic value of u, u_c is a characteristic size of the ra values, and t_c is a characteristic size of the range of t_c where u varies sign Choosing u_m, u_c, and t_c is not always easy and often an art in com problems. We just state one choice first:

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

nserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a nd b are constants, results after some algebra in the *scaled problem*

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u}, \quad \bar{u}(0) = 1 - \beta,$$

here β is a dimensionless number

$$\beta = \frac{b}{Ia}\,.$$

hat is, only the special combination of $b/(Ia)$ matters, not what the individual alues of b, a, and I are. Moreover, if $b = 0$, the scaled problem is independent f a and I! In practice this means that we can perform one numerical simulation f the scaled problem and recover the solution of any problem for a given a and by stretching the axis in the plot: $u = I\bar{u}$ and $t = \bar{t}/a$. For $b \neq 0$, we simulate ne scaled problem for a few β values and recover the physical solution u by anslating and stretching the u axis and stretching the t axis.

The scaling breaks down if $I = 0$. In that case we may choose $u_m = 0$, $_c = b/a$, and $t_c = 1/b$, resulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

s with $b = 0$, the case $I = 0$ has a scaled problem with no physical parameters!

It is common to drop the bars after scaling and write the scaled problem s $u' = -u$, $u(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementation of the roblem $u' = -au + b$, $u(0) = I$, can be reused for the scaled problem by setting $= 1$, $b = 0$, and $I = 1 - \beta$ in the code, if $I \neq 0$, or one sets $a = 1$, $b = 1$, nd $I = 0$ when the physical I is zero. Falling bodies in fluids, as described in ection 8.8, involves $u' = -au + b$ with seven physical parameters. All these anish in the scaled version of the problem if we start the motion from rest!

.2 Evolution of a population

et N be the number of individuals in a population occupying some spatial omain. Despite N being an integer in this problem, we shall compute with N s a real number and view $N(t)$ as a continuous function of time. The basic odel assumption is that in a time interval Δt the number of newcomers to the opulations (newborns) is proportional to N, with proportionality constant $\bar{b}$. he amount of newcomers will increase the population and result in to

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

is obvious that a long time interval Δt will result in more newcomers and ence a larger $\bar{b}$. Therefore, we introduce $b = \bar{b}/\Delta t$: the number of newcomers er unit time and per individual. We must then multiply b by the length of the

time interval considered and by the population size to get the total nu new individuals, $b\Delta tN$.

If the number of removals from the population (deaths) is also prop to N, with proportionality constant $d\Delta t$, the population evolves accord

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t)\,.$$

Dividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b - d)N, \quad N(0) = N_0\,.$$

In a population where the death rate (d) is larger than then newborn $a > 0$, and the population experiences exponential decay rather than exp growth.

In some populations there is an immigration of individuals into the domain. With I individuals coming in per time unit, the equation population change becomes

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t) + \Delta tI\,.$$

The corresponding ODE reads

$$N' = (b - d)N + I, \quad N(0) = N_0\,.$$

Some simplification arises if we introduce a fractional measure of the tion: $u = N/N_0$ and set $r = b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1,$$

where $f = I/N_0$ measures the net immigration per time unit as the fra the initial population. Very often, r is approximately constant, but f is a function of time.

The growth rate r of a population decreases if the environment has resources. Suppose the environment can sustain at most $N_{\max}$ individu may then assume that the growth rate approaches zero as N approache i.e., as u approaches $M = N_{\max}/N_0$. The simplest possible evolution then a linear function: $r(t) = r_0(1 - u(t)/M)$, where r_0 is the initial grov when the population is small relative to the maximum size and there is resources. Using this $r(t)$ in (108) results in the *logistic model* for the e of a population (assuming for the moment that $f = 0$):

$$u' = r_0(1 - u/M)u, \quad u(0) = 1\,.$$

Initially, u will grow at rate r_0, but the growth will decay as u approa and then there is no more change in u, causing $u \to M$ as $t \to \infty$. N the logistic equation $u' = r_0(1 - u/M)u$ is *nonlinear* because of the qu term $-u^2r_0/M$.

.3 Compound interest and inflation

ay the annual interest rate is r percent and that the bank adds the interest nce a year to your investment. If u^n is the investment in year n, the investment ı year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

ı reality, the interest rate is added every day. We therefore introduce a parameter ı for the number of periods per year when the interest is added. If n counts ıe periods, we have the fundamental model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,. \tag{110}$$

his model is a *difference equation*, but it can be transformed to a continuous ifferential equation through a limit process. The first step is to derive a formula ɔr the growth of the investment over a time t. Starting with an investment u^0, nd assuming that r is constant in time, we get

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right) u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

ıtroducing time t, which here is a real-numbered counter for years, we have ıat $n = mt$, so we can write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0\,.$$

he second step is to assume *continuous compounding*, meaning that the interest added continuously. This implies $m \to \infty$, and in the limit one gets the ɔrmula

$$u(t) = u_0 e^{rt/100}, \tag{111}$$

hich is nothing but the solution of the ODE problem

$$u' = \frac{r}{100}u, \quad u(0) = u_0\,. \tag{112}$$

his is then taken as the ODE model for compound interest if $r > 0$. However, ıe reasoning applies equally well to inflation, which is just the case $r < 0$. ɔne may also take the r in (112) as the net growth of an investemt, where r ıkes both compound interest and inflation into account. Note that for real pplications we must use a time-dependent r in (112).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is then a rocess exhibiting exponential decay according to

$$u' = -au, \quad u(0) = I\,.$$

8.4 Radioactive Decay

An atomic nucleus of an unstable atom may lose energy by emitting particles and thereby be transformed to a nucleus with a different nu protons and neutrons. This process is known as radioactive decay[33]. the process is stochastic when viewed for a single atom, because it is im to predict exactly when a particular atom emits a particle. Neverthele a large number of atoms, N, one may view the process as determini compute the mean behavior of the decay. Below we reason intuitivel an ODE for the mean behavior. Thereafter, we show mathematicall detailed stochastic model for single atoms leads the same behavi

Deterministic model. Suppose at time t, the number of the origin type is $N(t)$. A basic model assumption is that the transformation of th of the original type in a small time interval Δt is proportional to N, so

$$N(t + \Delta t) = N(t) - a\Delta t N(t),$$

where $a > 0$ is a constant. Introducing $u = N(t)/N(0)$, dividing by letting $\Delta t \to 0$ gives the following ODE:

$$u' = -au, \quad u(0) = 1\,.$$

The parameter a can for a given nucleus be expressed through the *half-* which is the time taken for the decay to reduce the initial amount by i.e., $u(t_{1/2}) = 0.5$. With $u(t) = e^{-at}$, we get $t_{1/2} = a^{-1}\ln 2$ or $a = \ln 2/$

Stochastic model. We have originally N_0 atoms. Each atom m decayed or survived at a particular time t. We want to count how many atoms that are left, i.e., how many atoms that have survived. The su a single atom at time t is a random event. Since there are only two ou survival or decay, we have a Bernoulli trial[34]. Let p be the probal survival (implying that the probability of decay is $1 - p$). If each atom independently of the others, and the probability of survival is the same f atom, we have N_0 statistically Bernoulli trials, known as a *binomial exp* from probability theory. The probability $P(N)$ that N out of the N_0 ato survived at time t is then given by the famous *binomial distribution*

$$P(N) = \frac{N_0!}{N!(N_0 - N)!}p^N(1-p)^{N_0-N}\,.$$

The mean (or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be $N_0 p$.

It remains to estimate p. Let the interval $[0, t]$ be divided into n subintervals of length Δt. We make the assumption that the probal decay of a single atom in an interval of length Δt is $\tilde{p}$, and that this pro

[33] http://en.wikipedia.org/wiki/Radioactive_decay
[34] http://en.wikipedia.org/wiki/Bernoulli_trial

proportional to Δt: $\tilde{p} = \lambda\Delta t$ (it sounds natural that the probability of ecay increases with Δt). The corresponding probability of survival is $1 - \lambda\Delta t$. elieving that λ is independent of time, we have, for each interval of length Δt, Bernoulli trial: the atom either survives or decays in that interval. Now, p ıould be the probability that the atom survives in all the intervals, i.e., that e have m successful Bernoulli trials in a row and therefore

$$p = (1 - \lambda\Delta t)^m \,.$$

he expected number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0 p = N_0(1 - \lambda\Delta t)^m, \quad m = t/\Delta t \,. \tag{114}$$

To see the relation between the two types of Bernoulli trials and the ODE bove, we go to the limit $\Delta t \to t$, $m \to \infty$. One can show that

$$p = \lim_{m\to\infty}(1 - \lambda\Delta t)^m = \lim_{m\to\infty}\left(1 - \lambda\frac{t}{m}\right)^m = e^{-\lambda t}$$

his is the famous exponential waiting time (or arrival time) distribution for Poisson process in probability theory (obtained here, as often done, as the mit of a binomial experiment). The probability of decay, $1 - e^{-\lambda t}$, follows an xponential distribution[35]. The limit means that m is very large, hence Δt is ery small, and $\tilde{p} = \lambda\Delta t$ is very small since the intensity of the events, λ, is ssumed finite. This situation corresponds to a very small probability that an tom will decay in a very short time interval, which is a reasonable model. The ame model occurs in lots of different applications, e.g., when waiting for a taxi, r when finding defects along a rope.

elation between stochastic and deterministic models. With $p = e^{-\lambda t}$ e get the expected number of original atoms at t as $N_0 p = N_0 e^{-\lambda t}$, which exactly the solution of the ODE model $N' = -\lambda N$. This gives also an nterpretation of a via λ or vice versa. Our important finding here is that the DE model captures the mean behavior of the underlying stochastic model. his is, however, not always the common relation between microscopic stochastic ıodels and macroscopic "averaged" models.

Also of interest is to see that a Forward Euler discretization of $N' = -\lambda N$, $(0) = N_0$, gives $N^m = N_0(1 - \lambda\Delta t)^m$ at time $t_m = m\Delta t$, which is exactly ıe expected value of the stochastic experiment with N_0 atoms and m small ıtervals of length Δt, where each atom can decay with probability $\lambda\Delta t$ in an ıterval.

A fundamental question is how accurate the ODE model is. The underlying ochastic model fluctuates around its expected value. A measure of the fluc- ıations is the standard deviation of the binomial experiment with N_0 atoms, hich can be shown to be $\mathrm{Std}[P] = \sqrt{N_0 p(1-p)}$. Compared to the size of the xpectation, we get the normalized standard deviation

[35] http://en.wikipedia.org/wiki/Exponential_distribution

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1} - 1} = N_0^{-1/2}\sqrt{(1 - e^{-\lambda t})^{-1} - 1} \approx (N_0\lambda t)^-$$

showing that the normalized fluctuations are very small if N_0 is very larg is usually the case.

8.5 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, ence shows that the temperature falls rapidly in the beginning, and t changes in temperature levels off until the body's temperature equals the surroundings. Newton carried out some experiments on cooling l and found that the temperature evolved as a "geometric progression at arithmetic progression", meaning that the temperature decayed expon Later, this result was formulated as a differential equation: the rate of cl the temperature in a body is proportional to the temperature difference the body and its surroundings. This statement is known as *Newton's cooling*, which can be mathematically expressed as

$$\frac{dT}{dt} = -k(T - T_s),$$

where T is the temperature of the body, T_s is the temperature of the surro t is time, and k is a positive constant. Equation (133) is primarily vi an empirical law, valid when heat is efficiently convected away from the of the body by a flowing fluid such as air at constant temperature T *heat transfer coefficient* k reflects the transfer of heat from the body surroundings and must be determined from physical experiments.

We must obviously have an initial condition $T(0) = T_0$ in additio cooling law (133).

8.6 Decay of atmospheric pressure with altitude

Vertical equilibrium of air in the atmosphere is governed by the equatic

$$\frac{dp}{dz} = -\varrho g \,.$$

Here, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807$ m standard value of the acceleration of gravity. (Equation (116) follows from the general Navier-Stokes equations for fluid motion, with the ass that the air does not move.)

The pressure is related to density and temperature through the ideal

$$\varrho = \frac{Mp}{R^*T},$$

here M is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal as constant (8.314 Nm/(mol K)), and T is the temperature. All variables p, ϱ, nd T vary with the height z. Inserting (117) in (116) results in an ODE with a ariable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,. \tag{118}$$

Iultiple atmospheric layers. The atmosphere can be approximately mod- ed by seven layers. In each layer, (118) is applied with a linear temperature of ıe form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

here $z = h_i$ denotes the bottom of layer number i, having temperature $\bar{T}_i$, and $_i$ is a constant in layer number i. The table below lists h_i (m), $\bar{T}_i$ (K), and L_i K/m) for the layers $i = 0, \ldots, 6$.

i	h_i	$\bar{T}_i$	L_i
0	0	288	-0.0065
1	11,000	216	0.0
2	20,000	216	0.001
3	32,000	228	0.0028
4	47,000	270	0.0
5	51,000	270	-0.0028
6	71,000	214	-0.002

or implementation it might be convenient to write (118) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p, \tag{119}$$

here $\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values given in ıe table. The value of the pressure at the sea level $z = 0$, $p_0 = p(0)$, is 101325 a.

implification: $L = 0$. One commonly used simplification is to assume that ıe temperature is constant within each layer. This means that $L = 0$.

implification: one-layer model. Another commonly used approximation to work with one layer instead of seven. This one-layer model[36] is based on $'(z) = T_0 - Lz$, with sea level standard temperature $T_0 = 288$ K and temperature ıpse rate $L = 0.0065$ K/m.

[36] http://en.wikipedia.org/wiki/Density_of_air

8.7 Compaction of sediments

Sediments, originally made from materials like sand and mud, get con through geological time by the weight of new material that is deposited sea bottom. The porosity ϕ of the sediments tells how much void (fluid there is between the sand and mud grains. The porosity reduces with because the weight of the sediments above and causes the void space t and thereby increase the compaction.

A typical assumption is that the change in ϕ at some depth z is ne proportional to ϕ. This assumption leads to the differential equation pr

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0,$$

where the z axis points downwards, $z = 0$ is the surface with known p and $c > 0$ is a constant.

The upper part of the Earth's crust consists of many geological layers on top of each other, as indicated in Figure 17. The model (120) can be for each layer. In layer number i, we have the unknown porosity functi fulfilling $\phi_i'(z) = -c_i z$, since the constant c in the model (120) dep the type of sediment in the layer. From the figure we see that new l sediments are deposited on top of older ones as time progresses. The com as measured by ϕ, is rapid in the beginning and then decreases (expon with depth in each layer.

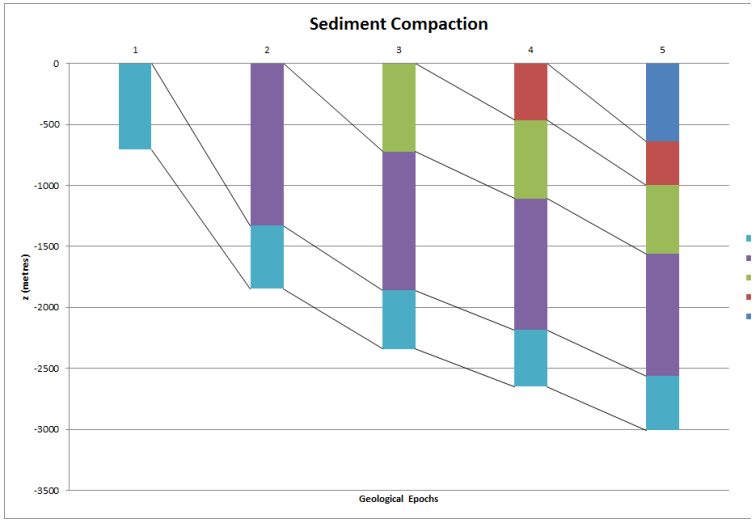


Figure 17: Illustration of the compaction of geological layers (with c colors) through time.

When we drill a well at present time through the right-most col sediments in Figure 17, we can measure the thickness of the sediment in (

ottom layer. Let L_1 be this thickness. Assuming that the volume of sediment emains constant through time, we have that the initial volume, $\int_0^{L_{1,0}} \phi_1 dz$, must qual the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is the depth of the bottom of ne sediment in the present day configuration. After having solved for ϕ_1 as a nction of z, we can then find the original thickness $L_{1,0}$ of the sediment from ne equation

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

n hydrocarbon exploration it is important to know $L_{1,0}$ and the compaction istory of the various layers of sediments.

.8 Vertical motion of a body in a viscous fluid

body moving vertically through a fluid (liquid or gas) is subject to three ifferent types of forces: the gravity force, the drag force[37], and the buoyancy rce.

verview of forces. The gravity force is $F_g = -mg$, where m is the mass f the body and g is the acceleration of gravity. The uplift or buoyancy force Archimedes force") is $F_b = \varrho g V$, where ϱ is the density of the fluid and V is ne volume of the body. Forces and other quantities are taken as positive in the pward direction.

The drag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu}, \tag{121}$$

here d is the diameter of the body in the direction perpendicular to the flow, v the velocity of the body, and μ is the dynamic viscosity of the fluid. When e < 1, the drag force is fairly well modeled by the so-called Stokes' drag, which r a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,. \tag{122}$$

or large Re, typically Re $> 10^3$, the drag force is quadratic in the velocity:

$$F_d^{(q)} = -\frac{1}{2} C_D \varrho A|v|v, \tag{123}$$

here C_D is a dimensionless drag coefficient depending on the body's shape, nd A is the cross-sectional area as produced by a cut plane, perpendicular to ne motion, through the thickest part of the body. The superscripts ^{q} and ^{S} in $^{(S)}_d$ and $F_d^{(q)}$ indicate Stokes drag and quadratic drag, respectively.

[37] http://en.wikipedia.org/wiki/Drag_(physics)

Equation of motion. All the mentioned forces act in the vertical d Newton's second law of motion applied to the body says that the sum forces must equal the mass of the body times its acceleration a in the direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

Here we have chosen to model the fluid resistance by the Stokes drag. I the expressions for the forces yields

$$ma = -mg - 3\pi d\mu v + \varrho g V\,.$$

The unknowns here are v and a, i.e., we have two unknowns but o equation. From kinematics in physics we know that the acceleration is t derivative of the velocity: $a = dv/dt$. This is our second equation. We ca eliminate a and get a single differential equation for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho g V\,.$$

A small rewrite of this equation is handy: We express m as $\varrho_b V$, where density of the body, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V} v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,.$$

We may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right),$$

so that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,.$$

The corresponding initial condition is $v(0) = v_0$ for some prescribed velocity v_0.

This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, to the result

$$v'(t) = -\frac{1}{2} C_D \frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,.$$

Defining

$$a = \frac{1}{2} C_D \frac{\varrho A}{\varrho_b V},$$

and b as above, we can write (127) as

$$v'(t) = -a|v|v + b\,.$$

'erminal velocity. An interesting aspect of (126) and (129) is whether v will pproach a final constant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A onstant v means that $v'(t) \to 0$ as $t \to \infty$ and therefore the terminal velocity $_T$ solves

$$0 = -av_T + b$$

nd

$$0 = -a|v_T|v_T + b\,.$$

he former equation implies $v_T = b/a$, while the latter has solutions $v_T = \sqrt{|b|/a}$ for a falling body ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

. Crank-Nicolson scheme. Both governing equations, the Stokes' drag iodel (126) and the quadratic drag model (129), can be readily solved by the orward Euler scheme. For higher accuracy one can use the Crank-Nicolson ıethod, but a straightforward application this method results a nonlinear quation in the new unknown value v^{n+1} when applied to (129):

$$\frac{v^{n+1}-v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,. \tag{130}$$

owever, instead of approximating the term $-|v|v$ by an arithmetic average, we an use a *geometric mean*:

$$(|v|v)^{n+\frac{1}{2}} \approx |v^n|v^{n+1}\,. \tag{131}$$

he error is of second order in Δt, just as for the arithmetic average and the entered finite difference approximation in (130). With this approximation trick, ıe discrete equation

$$\frac{v^{n+1}-v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

ecomes a *linear* equation in v^{n+1}, and we can therefore easily solve for v^{n+1}:

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,. \tag{132}$$

'hysical data. Suitable values of μ are $1.8\cdot 10^{-5}$ Pa s for air and $8.9\cdot 10^{-4}$ Pa s or water. Densities can be taken as 1.2 kg/m^3 for air and as $1.0\cdot 10^3$ kg/m^3 or water. For considerable vertical displacement in the atmosphere one should ake into account that the density of air varies with the altitude, see Section 8.6. 'ne possible density variation arises from the one-layer model in the mentioned ection.

Any density variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ in (132). o compute the density that enters $b^{n+\frac{1}{2}}$ we must also compute the vertical osition $z(t)$ of the body. Since $v = dz/dt$, we can use a centered difference pproximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

This $z^{n+\frac{1}{2}}$ is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then

The drag coefficient[38] C_D depends heavily on the shape of the bod values are: 0.45 for a sphere, 0.42 for a semi-sphere, 1.05 for a cube, 0. long cylinder (when the center axis is in the vertical direction), 0.75 for a 1.0-1.3 for a man in upright position, 1.3 for a flat plate perpendicula flow, and 0.04 for a streamlined, droplet-like body.

Verification. To verify the program, one may assume a heavy bod such that the F_b force can be neglected, and further assume a small such that the air resistance F_d can also be neglected. This can be obta setting μ and ϱ to zero. The motion then leads to the velocity $v(t) =$ which is linear in t and therefore should be reproduced to machine p (say tolerance 10^{-15}) by any implementation based on the Crank-Nic Forward Euler schemes.

Another verification, but not as powerful as the one above, can be b computing the terminal velocity and comparing with the exact expressio advantage of this verification is that we can also the test situation $\varrho \neq$

As always, the method of manufactured solutions can be applied to implementation of all terms in the governing equation, but the solution t no physical relevance in general.

Scaling. Applying scaling, as described in Section 8.1, will for the lin reduce the need to estimate values for seven parameters down to choos value of a single dimensionless parameter

$$\beta = \frac{\varrho_b g V\left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu I},$$

provided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled p $\bar{u}' = 1 - \bar{u}$, $\bar{u}(0) = 0$, has no need for estimating physical parameters. Thi that there is a single universal solution to the problem of a falling body from rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical cases correspond to stretchi axis and the $\bar{u}$ axis in this dimensionless solution. More precisely, the p velocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V\left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu}\bar{u}(t/(g(\varrho/\varrho_b - 1)))\,.$$

[38] `http://en.wikipedia.org/wiki/Drag_coefficient`

.9 Decay ODEs from solving a PDE by Fourier expansions

uppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

ith boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = (x)$. One may express the solution as

$$u(x,t) = \sum_{k=1}^{m} A_k(t) e^{ikx\pi/L},$$

or appropriate unknown functions A_k, $k = 1, \ldots, m$. We use the complex xponential $e^{ikx\pi/L}$ for easy algebra, but the physical u is taken as the real art of any complex expression. Note that the expansion in terms of $e^{ikx\pi/L}$ is ompatible with the boundary conditions: all functions $e^{ikx\pi/L}$ vanish for $x = 0$ nd $x = L$. Suppose we can express $I(x)$ as

$$I(x) = \sum_{k=1}^{m} I_k e^{ikx\pi/L}\,.$$

uch an expansion can be computed by well-known Fourier expansion techniques, ut the details are not important here. Also, suppose we can express the given (x,t) as

$$f(x,t) = \sum_{k=1}^{m} b_k(t) e^{ikx\pi/L}\,.$$

nserting the expansions for u and f in the differential equations demands that ll terms corresponding to a given k must be equal. The calculations results in ne follow system of ODEs:

$$A_k'(t) = -\alpha \frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1, \ldots, m\,.$$

rom the initial condition

$$u(x,0) = \sum_k A_k(0) e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

follows that $A_k(0) = I_k$, $k = 1, \ldots, m$. We then have m equations of the form $'_k = -aA_k + b$, $A_k(0) = I_k$, for appropriate definitions of a and b. These ODE roblems independent each other such that we can solve one problem at a time. he outline technique is a quite common approach for solving partial differential quations.

Remark. Since a_k depends on k and the stability of the Forward Euler demands $a_k \Delta t \leq 1$, we get that $\Delta t \leq \alpha^{-1} L^2 \pi^{-2} k^{-2}$. Usually, quite values are needed to accurately represent the given functions I and f and needs to be very small for these large values of k. Therefore, the Crank-N and Backward Euler schemes, which allow larger Δt without any growt solutions, are more popular choices when creating time-stepping algorit partial differential equations of the type considered in this example.

9 Exercises

Exercise 10: Derive schemes for Newton's law of coo

Show in detail how we can apply the ideas of the Forward Euler, Ba Euler, Crank-Nicolson, and θ-rule discretizations to derive explicit compu formulas for new temperature values in Newton's law of cooling (see Sect

$$\frac{dT}{dt} = -k(T - T_s), \quad T(0) = T_0\,.$$

Here, T is the temperature of the body, T_s is the temperature of the surro t is time, k is the heat transfer coefficient, and T_0 is the initial temper the body. Filename: `schemes_cooling.pdf`.

Exercise 11: Implement schemes for Newton's law o ing

Formulate a θ-rule for the three schemes in Exercise 10 such that you the three schemes from a single formula by varying the θ parameter. Im the θ scheme in a function `cooling(T0, k, T_s, t_end, dt, thet` where `T0` is the initial temperature, `k` is the heat transfer coefficient, `T_` temperature of the surroundings, `t_end` is the end time of the simulatio the time step, and `theta` corresponds to θ. The `cooling` function shoul the temperature as an array `T` of values at the mesh points and the time Construct verification examples to check that the implementation work

Hint. For verification, try to find an exact solution of the discrete eq A trick is to introduce $u = T - T_s$, observe that $u^n = (T_0 - T_s)A^n$ f amplification factor A, and then express this formula in terms of T^n. Filename: `cooling.py`.

Exercise 12: Find time of murder from body temper

A detective measures the temperature of a dead body to be 26.7 C at 2 p hour later the temperature is 25.8 C. The question is when death occu

Assume that Newton's law of cooling (133) is an appropriate mathe model for the evolution of the temperature in the body. First, dete

ı (133) by formulating a Forward Euler approximation with one time steep
om time 2 am to time 3 am, where knowing the two temperatures allows for
nding k. Assume the temperature in the air to be 20 C. Thereafter, simulate
ıe temperature evolution from the time of murder, taken as $t = 0$, when
$= 37$ C, until the temperature reaches 25.8 C. The corresponding time allows
ɔr answering when death occurred. Filename: `detective.py`.

Exercise 13: Simulate an oscillating cooling process

he surrounding temperature T_s in Newton's law of cooling (133) may vary in
me. Assume that the variations are periodic with period P and amplitude a
round a constant mean temperature T_m:

$$T_s(t) = T_m + a\sin\left(\frac{2\pi}{P}t\right). \tag{134}$$

imulate a process with the following data: $k = 20$ min^{-1}, $T(0) = 5$ C, $T_m = 25$
, $a = 2.5$ C, and $P = 1$ h. Also experiment with $P = 10$ min and $P = 3$ h.
lot T and T_s in the same plot. Filename: `osc_cooling.py`.

Exercise 14: Radioactive decay of Carbon-14

he Carbon-14[39] isotope, whose radioactive decay is used extensively in dating
rganic material that is tens of thousands of years old, has a half-life of 5,730
ears. Determine the age of an organic material that contains 8.4 percent of its
ıitial amount of Carbon-14. Use a time unit of 1 year in the computations. The
ncertainty in the half time of Carbon-14 is ±40 years. What is the corresponding
ncertainty in the estimate of the age?

Hint. Use simulations with $5,730 \pm 40$ y as input and find the corresponding
ıterval for the result.
ilename: `carbon14.py`.

Exercise 15: Simulate stochastic radioactive decay

he purpose of this exercise is to implement the stochastic model described in
ection 8.4 and show that its mean behavior approximates the solution of the
ɔrresponding ODE model.

The simulation goes on for a time interval $[0, T]$ divided into N_t intervals of
ngth Δt. We start with N_0 atoms. In some time interval, we have N atoms
ıat have survived. Simulate N Bernoulli trials with probability $\lambda\Delta t$ in this
ıterval by drawing N random numbers, each being 0 (survival) or 1 (decay),
here the probability of getting 1 is $\lambda\Delta t$. We are interested in the number
f decays, d, and the number of survived atoms in the next interval is then

[39] `http://en.wikipedia.org/wiki/Carbon-14`

$N - d$. The Bernoulli trials are simulated by drawing N uniformly dist
real numbers on $[0, 1]$ and saying that 1 corresponds to a value less tha

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
Bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = Bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true a
values become 1 and 0, respectively, when converted to an integer arra

Repeat the simulation over $[0, T]$ a large number of times, comp
average value of N in each interval, and compare with the solutior
corresponding ODE model. Filename: `stochastic_decay.py`.

Exercise 16: Radioactive decay of two substances

Consider two radioactive substances A and B. The nuclei in substance
to form nuclei of type B with a half-life $A_{1/2}$, while substance B decay
type A nuclei with a half-life $B_{1/2}$. Letting u_A and u_B be the fraction
initial amount of material in substance A and B, respectively, the fc
system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u_A' = u_B/B_{1/2} - u_A/A_{1/2},$$
$$\frac{1}{\ln 2}u_B' = u_A/A_{1/2} - u_B/B_{1/2},$$

with $u_A(0) = u_B(0) = 1$.

Make a simulation program that solves for $u_A(t)$ and $u_B(t)$. Ve
implementation by computing analytically the limiting values of u_A
as $t \to \infty$ (assume $u_A', u_B' \to 0$) and comparing these with those o
numerically.

Run the program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$
Use a time unit of 1 minute. Plot u_A and u_B versus time in the sa
Filename: `radioactive_decay_2subst.py`.

Exercise 17: Simulate the pressure drop in the atmos

We consider the models for atmospheric pressure in Section 8.6. Make a p
with three functions,

- one computing the pressure $p(z)$ using a seven-layer model and va
- one computing $p(z)$ using a seven-layer model, but with constant t
  ture in each layer, and
- one computing $p(z)$ based on the one-layer model.

:ow can these implementations be verified? Should ease of verification impact ow you code the functions? Compare the three models in a plot. Filename: `tmospheric_pressure.py`.

:xercise 18: Make a program for vertical motion in a fluid

nplement the Stokes' drag model (124) and the quadratic drag model (127) om Section 8.8, using the Crank-Nicolson scheme and a geometric mean for $|v|v$ as explained, and assume constant fluid density. At each time level, compute ıe Reynolds number Re and choose the Stokes' drag model if $\mathrm{Re} < 1$ and the uadratic drag model otherwise.

The computation of the numerical solution should take place either in a stand-lone function (as in Section 2.1) or in a solver class that looks up a problem ass for physical data (as in Section **??**). Create a module (see Section **??**) and quip it with nose tests (see Section **??**) for automatically verifying the code.

Verification tests can be based on

- the terminal velocity (see Section 8.8),
- the exact solution when the drag force is neglected (see Section 8.8),
- the method of manufactured solutions (see Section 5.5) combined with computing convergence rates (see Section **??**).

se, e.g., a quadratic polynomial for the velocity in the method of manufactured olutions. The expected error is $\mathcal{O}(\Delta t^2)$ from the centered finite difference pproximation and the geometric mean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the discrete quations in many problems. Show that this is true for linear drag (by adding source term that depends on t), but not for quadratic drag because of the eometric mean approximation. Use the method of manufactured solutions to dd a source term *in the discrete equations for quadratic drag* such that a linear unction of t is a solution. Add a nose test for checking that the linear function reproduced to machine precision in the case of both linear and quadratic drag.

Apply the software to a case where a ball rises in water. The buoyancy force here the driving force, but the drag will be significant and balance the other orces after a short time. A soccer ball has radius 11 cm and mass 0.43 kg. Start ıe motion from rest, set the density of water, ϱ, to 1000 kg/m^3, set the dynamic iscosity, μ, to 10^{-3} Pa s, and use a drag coefficient for a sphere: 0.45. Plot the elocity of the rising ball. Filename: `vertical_motion.py`.

roject 19: Simulate parachuting

he aim of this project is to develop a general solver for the vertical motion of a ody with quadratic air drag, verify the solver, apply the solver to a skydiver in ee fall, and finally apply the solver to a complete parachute jump.

All the pieces of software implemented in this project should be realized as ython functions and/or classes and collected in one module.

a) Set up the differential equation problem that governs the velocity motion. The parachute jumper is subject to the gravity force and a qu drag force. Assume constant density. Add an extra source term be u program verification. Identify the input data to the problem.

b) Make a Python module for computing the velocity of the motion. Als the module with functionality for plotting the velocity.

Hint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ to linearize the equation of motion with quadratic drag.

Hint 2. You can either use functions or classes for implementation. choose functions, make a function `solver` that takes all the input dat problem as arguments and that returns the velocity (as a mesh functi the time mesh. In case of a class-based implementation, introduce a p class with the physical data and a solver class with the numerical dat `solve` method that stores the velocity and the mesh in the class.

Allow for a time-dependent area and drag coefficient in the formula drag force.

c) Show that a linear function of t does not fulfill the discrete equations of the geometric mean approximation used for the quadratic drag te a source term, as in the method of manufactured solutions, such that function of t is a solution of the discrete equations. Make a nose test t that this solution is reproduced to machine precision.

d) The expected error in this problem goes like Δt^2 because we use a c finite difference approximation with error $\mathcal{O}(\Delta t^2)$ and a geometric m proximation with error $\mathcal{O}(\Delta t^2)$. Use the method of manufactured s combined with computing convergence rate to verify the code. Make a n for checking that the convergence rate is correct.

e) Compute the drag force, the gravity force, and the buoyancy for function of time. Create a plot with these three forces.

Hint. You can either make a function `forces(v, t, plot=None)` that the forces (as mesh functions) and `t` and shows a plot on the screen a saves the plot to a file with name `plot` if `plot` is not `None`, or you can the solver class with computation of forces and include plotting of force visualization class.

f) Compute the velocity of a skydiver in free fall before the parachute

Hint. Meade and Struthers [5] provide some data relevant to skydiving mass of the human body and equipment can be set to 100 kg. A sky spread-eagle formation has a cross-section of 0.5 m^2 in the horizontal pla

[40] `http://en.wikipedia.org/wiki/Parachuting`

ensity of air decreases varies altitude, but can be taken as constant, 1 kg/m^3, or altitudes relevant to skydiving (0-4000 m). The drag coefficient for a man in pright position can be set to 1.2. Start with a zero velocity. A free fall typically as a terminating velocity of 45 m/s. (This value can be used to tune other arameters.)

) The next task is to simulate a parachute jumper during free fall and after ıe parachute opens. At time t_p, the parachute opens and the drag coefficient ıd the cross-sectional area change dramatically. Use the program to simulate a ump from $z = 3000$ m to the ground $z = 0$. What is the maximum acceleration, ıeasured in units of g, experienced by the jumper?

lint. Following Meade and Struthers [5], one can set the cross-section area erpendicular to the motion to 44 m^2 when the parachute is open. Assume ıat it takes 8 s to increase the area linearly from the original to the final value. he drag coefficient for an open parachute can be taken as 1.8, but tuned using ıe known value of the typical terminating velocity reached before landing: 5.3 ı/s. One can take the drag coefficient as a piecewise constant function with ı abrupt change at t_p. The parachute is typically released after $t_p = 60$ s, but ırger values of t_p can be used to make plots more illustrative.
ilename: `skydiving.py`.

Exercise 20: Formulate vertical motion in the atmosphere

ertical motion of a body in the atmosphere needs to take into account a varying ir density if the range of altitudes is many kilometers. In this case, ϱ varies with ıe altitude z. The equation of motion for the body is given in Section 8.8. Let s assume quadratic drag force (otherwise the body has to be very, very small). differential equation problem for the air density, based on the information for ıe one-layer atmospheric model in Section 8.6, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p, \tag{137}$$

$$\varrho = p\frac{M}{R^*T}\,. \tag{138}$$

o evaluate $p(z)$ we need the altitude z. From the principle that the velocity is ıe derivative of the position we have that

$$z'(t) = v(t), \tag{139}$$

here v is the velocity of the body.

Explain in detail how the governing equations can be discretized by the For- ard Euler and the Crank-Nicolson methods. Filename: `falling_in_variable_dens`

Exercise 21: Simulate vertical motion in the atmospl

Implement the Forward Euler or the Crank-Nicolson scheme derived cise 20. Demonstrate the effect of air density variation on a falling hum the famous fall of Felix Baumgartner[41]. The drag coefficient can be set

Remark. In the Crank-Nicolson scheme one must solve a 3×3 sy equations at each time level, since p, ϱ, and v are coupled, while each e can be stepped forward at a time with the Forward Euler scheme. Fi `falling_in_variable_density.py`.

Exercise 22: Compute $y = |x|$ by solving an ODE

Consider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

which has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = x_2 = 1$, calculate by hand y_1 and y_2 from the Forward Euler, Backwar Crank-Nicolson, and Leapfrog methods. Use all of the former three met computing the y_1 value to be used in the Leapfrog calculation of y_2. Th visualize how these schemes perform for a uniformly partitioned me $N = 10$ and $N = 11$ points. Filename: `signum.py`.

Exercise 23: Simulate growth of a fortune with ra interest rate

The goal of this exercise is to compute the value of a fortune subject to i and a random interest rate. Suppose that the inflation is constant at i per year and that the annual interest rate, p, changes randomly at ea step, starting at some value p_0 at $t = 0$. The random change is from a v at $t = t_n$ to $p_n + \Delta p$ with probability 0.25 and $p_n - \Delta p$ with probabil No change occurs with probability 0.5. There is also no change if p^{n+1} 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial scaled to 1, and simulate 1000 scenarios of length 20 years. Compute tl evolution of one unit of money and the corresponding standard deviatic the mean curve along with the mean plus one standard deviation and tl minus one standard deviation. This will illustrate the uncertainty in tl curve.

Hint 1. The following code snippet computes p^{n+1}:

[41] `http://en.wikipedia.org/wiki/Felix_Baumgartner`

```
import random

def new_interest_rate(p_n, dp=0.5):
    r = random.random()  # uniformly distr. random number in [0,1)
    if 0 <= r < 0.25:
        p_np1 = p_n + dp
    elif 0.25 <= r < 0.5:
        p_np1 = p_n - dp
    else:
        p_np1 = p_n
    return (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

Hint 2. If $u_i(t)$ is the value of the fortune in experiment number i, $i = , \ldots, N-1$, the mean evolution of the fortune is

$$\bar{u}(t) = \frac{1}{N}\sum_{i=0}^{N-1} u_i(t),$$

nd the standard deviation is

$$s(t) = \sqrt{\frac{1}{N-1}\left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1}(u_i(t))^2\right)}.$$

uppose $u_i(t)$ is stored in an array `u`. The mean and the standard deviation f the fortune is most efficiently computed by using two accumulation arrays, `um_u` and `sum_u2`, and performing `sum_u += u` and `sum_u2 += u**2` after every xperiment. This technique avoids storing all the $u_i(t)$ time series for computing ıe statistics.
ilename: `random_interest.py`.

Exercise 24: Simulate a population in a changing environment

We shall study a population modeled by (108) where the environment, represented y r and f, undergoes changes with time.

) Assume that there is a sudden drop (increase) in the birth (death) rate at me $t = t_r$, because of limited nutrition or food supply:

$$a(t) = \begin{cases} r_0, & t < t_r, \\ r_0 - A, & t \geq t_r, \end{cases}$$

his drop in population growth is compensated by a sudden net immigration at me $t_f > t_r$:

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

tart with r_0 and make $A > r_0$. Experiment with these and other parameters ɔ illustrate the interplay of growth and decay in such a problem. Filename: `opulation_drop.py`.

b) Now we assume that the environmental conditions changes periodica time so that we may take

$$r(t) = r_0 + A\sin\left(\frac{2\pi}{P}t\right).$$

That is, the combined birth and death rate oscillates around r_0 with a m change of $\pm A$ repeating over a period of length P in time. Set f = experiment with the other parameters to illustrate typical features of the Filename: `population_osc.py`.

Exercise 25: Simulate logistic growth

Solve the logistic ODE (109) using a Crank-Nicolson scheme where (u' approximated by a *geometric mean*:

$$(u^{n+\frac{1}{2}})^2 \approx u^{n+1}u^n.$$

This trick makes the discrete equation linear in u^{n+1}. Filename: `logistic`

Exercise 26: Rederive the equation for continuous pound interest

The ODE model (112) was derived under the assumption that r was c Perform an alternative derivation without this assumption: 1) start wit 2) introduce a time step Δt instead of m: $\Delta t = 1/m$ if t is meas years; 3) divide by Δt and take the limit $\Delta t \to 0$. Simulate a case wl inflation is at a constant level I percent per year and the interest rate o $r = -I/2 + r_0\sin(2\pi t)$. Compare solutions for $r_0 = I, 3I/2, 2I$. Fi `interest_modeling.py`.

References

[1] D. Griffiths, F. David, and D. J. Higham. *Numerical Methods for C Differential Equations: Initial Value Problems.* Springer, 2010.

[2] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Diff Equations I. Nonstiff Problems.* Springer, 1993.

[3] G. Hairer and E. Wanner. *Solving Ordinary Differential Equa* Springer, 2010.

[4] H. P. Langtangen. *A Primer on Scientific Programming With Pytho* in Computational Science and Engineering. Springer, third edition,

[5] D. B. Meade and A. A. Struthers. Differential equations in the new mi the parachute problem. *International Journal of Engineering Ed* 15(6):417–424, 1999.

i] L. Petzold and U. M. Ascher. *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, volume 61. SIAM, 1998.

Index